Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Henri Hagberg

Performance of serialized object processing in Java

Master's Thesis
Espoo, July 29, 2019

Supervisor: Senior University Lecturer Vesa Hirvisalo
Advisor: Lic.Sc. Tapio Pitkäranta

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Henri Hagberg		
Title:	Performance of serialized object processing in Java		
Date:	July 29, 2019	**Pages:**	85
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Senior University Lecturer Vesa Hirvisalo		
Advisor:	Lic.Sc. Tapio Pitkäranta		

Distributed systems have become increasingly more common. In these systems, multiple nodes communicate with each other, typically over a network, which requires serialization of in-memory objects to byte streams. Object serialization has many uses also in more traditional systems. For example, serialization can be used for data persistence. As serialization in many systems is a central and frequent operation, its performance is critical for the whole application.

This thesis compares the performance of different Java serialization solutions in the context of a distributed database system called *FastormDB*. Transaction management in the system is centralized meaning that all nodes must transfer their changes over network to the central transactor node for committing. This requires object serialization and as the amount of change data can be tens of gigabytes, the performance of the chosen serialization solution is critical. We also evaluate several compression methods together with the serializers.

Serializers and compressors were first compared using synthetic benchmarks where considerable differences between serializers were found. However, in a throughput benchmark which simulates realistic use of the system the differences were insignificant. Usage of compression was also found to lower throughput despite relatively low network bandwidth. These results are partially explained by the current implementation of the system. This thesis identified some areas of improvement in the system and suggests that the most promising benchmarked technologies are re-evaluated when the improvements have been applied.

Keywords:	Java, object serialization, compression, distributed systems
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Henri Hagberg

Työn nimi:
Serialisoitujen objektien prosessoinnin suorituskyky Javassa

Päiväys:	29. heinäkuuta 2019	**Sivumäärä:**	85
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Vanhempi yliopistonlehtori Vesa Hirvisalo		
Ohjaaja:	Tekniikan lisensiaatti Tapio Pitkäranta		

Hajautetut järjestelmät ovat nykypäivänä yhä yleisempiä. Näissä järjestelmissä useat noodit kommunikoivat keskenään, tyypillisesti verkkoyhteyden yli, mikä vaatii koneen muistissa olevien objektien serialisointia tavuvirroiksi. Objektien serialisointia käytetään myös perinteisemmissä järjestelmissä. Serialisointia voidaan käyttää esimerkiksi datan tallentamiseen. Koska serialisointi on monissa järjestelmissä keskeinen ja toistuva toiminto, sen suorituskyky on kriittinen koko järjestelmälle.

Tässä diplomityössä vertaillaan eri Java-ohjelmointikielen serialisointiratkaisujen suorituskykyä *FastormDB*-nimisen hajautetun tietokannan yhteydessä. Järjestelmässä transaktionhallinta on toteutettu keskitetysti, mikä tarkoittaa, että noodien täytyy lähettää muutoksensa transaktorinoodille tietokantaan tallennettavaksi. Tämä edellyttää objektien serialisointia ja koska muutosten määrä voi olla jopa kymmeniä gigatavuja, on valitun serialisointiratkaisun suorityskyky ensiarvoisen tärkeää. Työssä myös vertaillaan eri pakkausmenetelmiä yhdessä serialisoijien kanssa.

Serialisointi- ja pakkausmenetelmiä verrattiin ensin synteettisissä suorituskykytesteissä, joissa löydettiin huomattavia eroja serialisoijien välillä. Toisaalta läpisyöttötestissä, joka simuloi järjestelmän realistista käyttöä, erot olivat merkityksettömiä. Datan pakkaamisen myös havaittiin huonontavan suoritustehoa verkkoyhteyden melko pienestä kaistanleveydestä huolimatta. Tulokset on osin selitettävissä järjestelmän nykyisellä toteutustavalla. Diplomityössä tunnistettiin joitakin kehitysalueita järjestelmässä ja ehdotetaan lupaavimpien vertailtujen teknologioiden uudelleenarviointia kun parannukset on toteutettu.

Asiasanat:	Java, objektien serialisointi, pakkaus, hajautetut järjestelmät
Kieli:	Englanti

Acknowledgements

I wish to thank my supervisor Vesa Hirvisalo and advisor Tapio Pitkäranta for their guidance and insights throughout the thesis process. You helped me keep this project in flight and in correct course. I am also grateful for the support and advice of my colleagues at RELEX Solutions, and for being awesome people in general. Lastly, I wish to thank my family and friends for motivating me and keeping my spirits high over this endeavor.

Espoo, July 29, 2019

Henri Hagberg

Abbreviations and Acronyms

API	Application Programming Interface
DBMS	Database Management System
JIT	Just-In-Time
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
RPC	Remote Procedure Call
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

Object serialization is an operation that is performed when data in the main memory of a computer has to be persisted or transferred to another computer. Serialization is commonly done in communication between web service frontend and backend or remote procedure calls between different services. Serialization can incur considerable overhead on any operation where it is used and therefore the performance of serialization technologies has been the subject of multiple studies [2, 10, 35, 46, 57]. However, these studies mostly consider objects with sizes of some kilobytes at most due to that being the size in many practical use cases, such as remote procedure calls (RPC). Nevertheless, there are cases where objects can be considerably larger.

1.1 Problem statement

The case company has a proprietary database system which is relatively tightly coupled to a business logic engine. Recently, there have been efforts to make the system distributed to improve both fault-tolerance and performance. The design features centralized transaction management: individual nodes can work on their private copy of the database but all changes must be committed via a central transactor node.

Currently, the system uses Java's built-in serialization to serialize change data that is moved between nodes. As there is evidence that Java serialization is slow, there have been concerns that using it might have a considerable negative on overall system throughput, especially considering the large amount of data the system handles. Additionally, data is not compressed in transit and it is unclear which, if any, compression method provides the best throughput.

1.2 Thesis contribution

This thesis evaluates the performance of different Java serialization libraries. The context of evaluation is a distributed database system, called *FastormDB*. We also consider compression together with serialization.

First, the author surveys the state of object serialization and selects Java serialization methods to compare based on the presented criteria. Several suitable compression methods are also selected for comparison in combination with the serializers. The author develops a workload generation method and synthetic benchmarks to compare serialization and deserialization performance of serializer-compressor combinations.

Second, to contrast the results gained from synthetic benchmarks, the author performs a throughput benchmark which measures the system throughput in realistic conditions. The author modifies the system to enable testing of different serializers and compressors and using the workload generation method used in the synthetic benchmarks. No other changes are done to the system as part of this thesis.

The overall goal of these benchmarks is to find the serializer-compressor combination which achieves the highest throughput. That is, the aim is to minimize the time from initiation of commit protocol to finish. As the number of evaluated combinations is already rather large, most of the libraries are tested with default settings.

We find that there are major performance differences between serializers in the synthetic benchmarks. Based on these results, it is seemingly beneficial to use a third-party serializer instead of Java serializer. Compression methods also have clear differences although it is not immediately evident which provides the best throughput.

However, in the throughput benchmark performance differences between serializers are insignificant and for throughput, it is best to not use compression. During the analysis of the results, we discover potential places of improvement in the system. We also discover some issues with the default configuration of one tested serializer and compressor. Regardless, we identify a serializer and a compressor that should be re-evaluated when the case company develops the system further.

1.3 Thesis structure

The rest of this thesis is organized as follows. First, an overview of databases and distributed systems is given in chapter 2. Object serialization and its state in the literature is discussed in chapter 3. In chapter 4 we describe

the system that is being measured. Evaluated technologies are presented in chapter 5 and performed benchmarks in chapter 6. Results are presented in chapter 7 and their analysis in chapter 8. Chapter 9 concludes the thesis.

Chapter 2

Databases and Distributed Systems

This chapter gives background information about databases and distributed systems. Although the practical contribution of this thesis concentrates on serialization, one should know the basics about databases and distributed systems to understand why the system considered in this thesis is designed as it is.

2.1 Databases

Databases are a major area in computer science and as such, there is plenty of literature on the subject. This section is based on one textbook about the topic: A First Course in Database Systems by Ullman and Widow [55].

2.1.1 Database management systems

A database is defined as a collection of information which lives over a long period. A database is managed by a database management system (DBMS). Although the definition distinguishes database from the system managing it, in common parlance *database* often refers to the combination of database and DBMS. This convention is also followed in this thesis.

A database management system is usually expected to have at least the following features:

1. Ability to create new databases with a specified schema.
2. Ability to query and modify the database.

3. Support storage and manipulation of large amounts of data efficiently.
4. Fault tolerance meaning stored data is not lost in case of failures.
5. Parallel access to data is supported without interfering with other users and no partial or inconsistent modifications to data are done.

To enable the last item, databases support *transactions* which are generally expected to have so-called ACID properties. These are discussed next.

2.1.2 Transactions and ACID properties

A database transaction is a group of operations which are executed atomically, or in other words, either all the operations are applied to the database or none are. If a fault occurs during a transaction, the database ensures that any changes already applied are reverted. This property is called *atomicity*.

On the other hand, if the transaction completes successfully (it is said to be committed), the database is expected to guarantee that the changes made by the transaction are persisted even in case of future failures. This is known as *durability*.

Databases typically allow the user to define various constraints. Examples of such constraints are that values must be non-null or that all references must point to existing rows. Transactions should respect these constraints and if any constraints would be violated by commit, the transaction should be aborted. This is known as *consistency*. Another way of defining consistency is that a consistent transaction maps a valid database state to another valid state.

The final ACID property is called *isolation*. As mentioned in the previous section, it is expected that multiple users can access the database without interfering with each other. Consequently, there can be multiple ongoing transactions but modifications done by those transactions should not be visible until they have committed. In the strictest sense, although transactions T_1 and T_2 might run in parallel, their combined effect is as if T_1 was executed first and T_2 after it. This is called *serializability*.

Although these definitions provide rather strong guarantees about the behavior of transactions, databases in practice offer more relaxed guarantees. For example, transactions might not be serializable as that often requires excessive locking which is detrimental for performance. Isolation level, in particular, is usually configurable.

2.1.3 Relational data model

A *data model* describes how data is organized in a database and what operations and constraints on data are possible. Perhaps the most widely used is the relational data model which represents data as two-dimensional tables called *relations*. Columns of a relation, which have name and type, are called *attributes*. The name of a relation and its attributes form the *schema* of a relation and a set of schemas form a *relational database schema*.

Rows of a relation are called *tuples* and each tuple contains one value for each attribute of the relation. One or more attributes are typically declared *keys*. The combination of the key values must be unique in a relation, imposing a *constraint* on the relation. By definition, keys uniquely identify tuples. Keys are used as identifiers when a tuple refers another tuple in same or different relation.

Interaction with relations is done using queries which include operations such as insertion or selection of tuples. Queries are typically written in some special query language such as the aptly named Structured Query Language (SQL). Query languages are usually less powerful compared to general-purpose programming languages. However, this is a strength as the DBMS can optimize queries more easily when expressiveness of the language is limited.

Data models only describe the logical organization of data and they don't impose requirements on the physical layout. For example, any relational database must have some method of mapping two-dimensional data (relations) to one dimension (computer memory). Although an implementation detail to some degree, physical layout greatly affects the properties of a database. This topic is discussed further in section 4.1.

2.2 Distributed databases

As discussed in the previous section, databases are expected to handle large amounts of data efficiently. As the amount of data grows, a natural coping strategy is to distribute the workload to multiple computers. In this section, we discuss distributed databases and distributed systems in general and some of the challenges faced in their implementation.

2.2.1 Distributed systems

A distributed system can be defined as "a collection of autonomous computing elements that appears to its users as a single coherent system" [56]. This

definition encapsulates two important characteristics. Firstly, they are collections of elements (henceforth called *nodes*) operating autonomously. For example, by default, there is no global clock which would allow nodes to agree on the order of events. Nodes may also have different roles and responsibilities which raises the question of how those are coordinated and how the membership of a node in the system is decided to begin with. Communication between nodes is done over network links which increases latency and constrains the amount of data that can be transferred between nodes.

Secondly, distributed systems should appear as a single, coherent system to the user. For example, the result of a query should not depend on the node that handles it. Ideally, for the user, a distributed system is indistinguishable from a non-distributed one. Achieving this is difficult in practice.

Two important motivations for building a distributed system are scalability and fault tolerance [28]. In a horizontally scalable distributed system it is possible to simply add more nodes to the system to counter an increased load. Vertical scaling, i.e. using a more powerful machine, is an option in many cases but at some point this becomes very expensive or outright impossible.

An important concept regarding distributed systems is the notion of partial failures [56]. In a non-distributed system, a machine failure means that service becomes unavailable. On the other hand, in a distributed system other nodes can continue operation while taking over responsibilities of a failed node. However, fault tolerance is not an inherent property of distributed systems. Rather, it is a property that is enabled only by careful design.

2.2.2 Promises and challenges of distributed databases

When it comes to the design of a distributed database, a central decision is how to logically and physically distribute the data among the nodes. There are two approaches which are orthogonal to some degree. The first approach is to store different relations on different nodes. This is called *partitioning* [44]. Individual relations can be also partitioned, either in row-wise (horizontal partitioning) or column-wise (vertical partitioning) manner. Partitioning can improve performance as queries targeting different relations or subsets of relations can be run on different nodes. Sufficiently large relations might not even fit on a single node. Partitioning can alleviate this.

Besides partitioning, distributed databases can also store the same data on multiple nodes. This is called *replication* [44]. Replication improves fault-tolerance of the system as in a situation where one node fails others can still respond to queries due to having the same data. Replication can potentially also improve performance: different queries targeting the same data can

be served by different nodes and geographically distributing data improves access times.

One of the most important features of any database is that stored data is consistent and durable. As mentioned, data replication improves durability. However, naïve replication causes issues with consistency: data items might have different values on different nodes as updates are not propagated instantaneously and the result of a query depends on the node that handles it [56]. Likewise, if a transaction modifies partitions on different nodes, a failure of one node easily leads to inconsistent data unless the system is specifically designed to tolerate it.

Consistency of data can be divided into two separate issues [44]. The first one, known as *mutual consistency*, means that nodes agree on the value of a data item. Distributed databases typically replicate data to some degree to improve fault tolerance. However, this raises the question about how and where to update replicated valued. There are algorithms, such as *two-phase commit* [17], which ensure that values are updated atomically on all replicas. This achieves *strong consistency* which means that all replicas agree on values of all data items at all times.

Strongly consistent systems require more coordination which typically translates to decreased performance. On the other hand, sometimes it is acceptable that committing transaction does not update values on all replicas. In these cases, it is expected that updates are propagated to other nodes at some point in the future. This is known as *eventual consistency* as the database is inconsistent for some period of time but reaches consistent state eventually [56].

Mutual consistency says that data items have equal values over the system but by itself that does not guarantee that those values are correct. To guarantee correctness, the system should also have *transactional consistency* [44]. Transactional consistency means that the execution history of transactions is globally serializable and transactions are properly isolated (in the sense defined in section 2.1.2). Without transactional consistency, two transactions could be executed in a different order on different nodes and their results partially overwriting each other while updates are propagated. In this case values would be consistent but incorrect.

2.2.3 CAP theorem

Challenges of maintaining data consistency were discussed in the previous section. In a perfect world where faults do not happen, consistency is a solved problem. However, in practice distributed system designs must assume that servers will crash and connections get cut. In addition to consistency,

systems should also maintain some degree of availability in the face of failures. Relationship between these properties is condensed in the following theorem [21]:

CAP theorem. *A distributed system may have at most two of the three following properties:*

- ***Consistency****: shared and replicated data appears same across the system.*
- ***Availability****: system eventually makes progress in a given task (e.g. updating a value of data item).*
- ***Partition tolerance****: system tolerates partitioning of its nodes (e.g. due to network failure).*

The practical implication of CAP theorem is that distributed systems must choose either availability or consistency since partitions cannot be avoided [21]. Which to choose depends on the application. Returning any answer at all might be more important for a content cache than returning an exactly accurate answer. In banking, for example, consistency overrides any other concern.

Chapter 3

Object Serialization

Serialization, in the context of programming[1], means "persist[ing] in-memory data by converting [objects] to a binary, text, or some other representation" [38]. Major use cases for serialization are persistence (e.g. writing data to file) and communication (e.g. sending message to another node over the network).

Broadly speaking, any non-trivial program manipulates data as a collection of objects which refer to each other using some pointer-like mechanism. As these point to addresses in memory, a valid pointer in one execution of the program is invalid in another. Moreover, the representation of primitive values is hardware dependent so even primitive data cannot be persisted or copied between processes in the general case. For these reasons there needs to be a conversion step which is known as serialization. The reverse operation, conversion from bytes back to objects, is called deserialization.

3.1 Serialization formats

There are many different serialization formats with different strengths and weaknesses. Next, we present one categorization of formats. This categorization also forms the basis of selection criteria presented in chapter 5.

3.1.1 Language-specific formats

Built-in support for object serialization is commonly offered by programming languages [41, 47, 52]. These are convenient for the programmer but consumers of serialized data are forced to use the same language. Evolution of

[1] Object serialization should not be confused with serializability of transactions.

classes, e.g. removing fields, might be difficult and even compatibility of serialized data between different versions of the runtime is not always guaranteed [52]. These are issues especially if serialization is used for data persistence. However, for more transient use cases (e.g. data is immediately discarded after processing), these limitations can be acceptable.

Built-in serialization is also a common source of vulnerabilities due to the ability to instantiate arbitrary classes [14, 60]. There are plans to remove Java built-in serialization altogether due to the number of security issues it has caused [42].

3.1.2 Text-based and binary formats

Some serialization formats, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language), represent data as text [28]. An obvious benefit is that the data is human-readable which simplifies troubleshooting and allows users to modify data without specialized tools. The downside of these formats is verbosity as fields have to be delimited with commas and quotes, for example, and additional whitespace is included to improve readability. Additionally, raw binary data, such as images, cannot be necessarily represented as text without additional encoding [28].

Binary serialization formats encode data using a format-specific binary encoding scheme. These formats offer a smaller footprint than textual formats. For example, number 2 147 483 647 can be represented with 4 bytes in binary but it takes 10 bytes using common character encoding schemes. Instead of using delimiters, field length is usually stored separately or inferred from the data type [37, 50]. Binary data is also often faster to parse as in the optimal case the serialized format can be the same as the in-memory format [4].

3.1.3 Use of schema

Schema is a description of serialized data which typically lists at least field names and types. It can include other metadata as well: for example, it might be possible to mark a field as optional or deprecated. Protocol Buffers is an example of a format that requires a schema [50]. Schema is not necessarily included with serialized data so consumers must acquire it separately. However, this approach can considerably reduce the amount of transferred data in use cases such as RPC where messages with the same format are sent repeatedly [28]. Other formats, such as Apache Avro, include a copy of the schema with the data [5]. This can still yield space savings if the same structure (e.g. array of objects) appears multiple times in the data.

Besides space savings, schemas have other benefits [28]. Schemas serve as always up-to-date documentation for the serialized data and can be used to validate incoming data. Since modifying schemas is always an explicit action, schemas also help evolving messages used in the communication of e.g. a distributed system in a controlled manner. Finally, schemas can be often used to generate code for reading and writing serialized data [23].

Downsides of schemas are that they must be always available when data is handled and all data must be strictly conforming. The alternative is schemaless formats. These formats, also called self-describing, embed field names and other metadata, such as data types, in the data itself. This is perhaps one reason why schemaless formats such as JSON and XML are commonly used for data interchange. Some schemaless formats also support optional schema for data validation [27, 58]. However, even in the absence of an explicit schema, code handling data usually makes some assumptions which impose an implicit schema on the data.

3.2 Object serialization in literature

Performance of Java serialization has been the subject of many studies. In many cases serialization has been evaluated as part of remote method invocation (RMI) [2, 12, 35, 46]. The common conclusion has been that RMI is too slow for many practical uses cases and a large part of slowness is caused by serialization. Improvements have been suggested, ranging from drop-in replacements [46] to libraries which can leverage high-performance network hardware [10]. Benchmarks in these studies have typically used small objects, sized at several kilobytes at most, as inputs.

Distributed data processing systems, such as Apache Spark [8], have created new kind of need for study and improvement of object serialization. Such systems use serialization when caching data to disk or shuffling it over the network to other nodes for processing. Intuition would tell that for workloads involving I/O and large amounts of data it is useful to spend CPU time to minimize the amount of data using efficient serialization techniques or compression. However, research has shown the gains are not always nearly as much as expected and minimizing I/O at the expense of CPU cycles can be even detrimental for the overall throughput. Ousterhout et al. [43] found that eliminating all waiting for disk and network I/O would improve performance at most 19 % and 2 %, respectively, for two big data benchmarks on Apache Spark. Li et al. [30] measured that the effect of using serialization and compression to avoid I/O ranges from a slowdown of 56 % to a speedup of 35 % on a variety of workloads on Apache Spark.

Performance of different Java and C++ serialization implementations formats in presence of network and disk I/O was studied by Sikdar et al. [53] and client protocols of widely used database management systems were similarly compared by Raasveldt et al. [51]. Both studies concluded that even considering I/O overhead, serialization, in general, is expensive enough that minimizing time spent on serialization is usually better than minimizing the size of serialized data. However, Sikdar et al. [53] observed that although the difference between fastest and second fastest implementation was up to an order of magnitude, as the amount of data increased the difference practically disappeared. Moreover, Raasveldt et al. [51] concluded that size-efficient serialization methods and use of compression yield better throughput with low-bandwidth and high-latency networks. These results strongly imply that there is a tipping point when I/O overhead starts to dominate CPU overhead and its exact position should be measured case-by-case.

As serialization cannot be always avoided, there have been attempts to reduce its performance impact. Horváth et al. [24] used code generation to improve the performance of Apache Flink [6]. Flink uses serialization for a variety of tasks such as fault-tolerance and, interestingly, it can perform some operations directly on serialized data. In the study, analysis of Java serialization attributed its slowness to use of reflection and the fact that the same mechanism is used to serialize all classes which results in code difficult for JIT compiler to optimize. Also, there is no special handling for primitive data types which adds overhead such as unnecessary boxing. By dynamically generating and compiling serializers for different classes on-demand, serialization performance was improved by a factor of six and overall Flink performance by 20 %. Dynamic code generation is also used by Gligoric et al. [22] to make deserialization as fast as possible for use cases where the same object is deserialized multiple times. Instead of generating serializers dynamically, the approach generates code which directly reconstructs the object. This yields 2 to 11 times faster deserialization speed compared to standard Java serialization.

Another interesting approach is taken by Apache Arrow [4]. Arrow specifies a language-independent memory format so applications using Arrow don't have to spend time on serialization and deserialization when exchanging data. The format also supports zero-copy reads, that is, applications can receive Arrow data to a buffer and use that data without any additional conversion. However, to leverage these features Arrow data structures need to be supported throughout the application. Although such in-place data structures have a larger memory footprint than efficient serialization formats, it is possible to eliminate CPU overhead almost completely [53].

Chapter 4

System Description

This thesis evaluates serialized object processing in the context of a specific distributed system. The system is described in this chapter.

4.1 FastormDB

FastormDB is a proprietary database developed by the case company, RELEX Solutions. Its key features are columnar layout and capability of in-memory computation [19]. Many relational databases lay out data in row-wise order meaning that values of a single row are next to each other in memory or on disk. In columnar layout, however, values of a single column are written in one run. Columnar layout has typically better compressibility compared to row layout as values of a single column have the same type and usually similar values. For queries which target only a few columns, columnar layout also offers better performance compared to row layout [28].

In-memory computation means that, although data is also persisted on disk, FastormDB keeps most of the data always in random access memory. Keeping data in memory vastly decreases latency compared to reading it from disk, which is the approach many traditional databases take. The downside is increased hardware requirements. FastormDB's in-memory computation is enabled by aggressive compression that is possible due to the columnar layout.

General-purpose databases are often deployed in a client-server setup where the database and the application using the database run in separate processes, possibly on separate machines. This model adds overhead as all data required by the application must be serialized and sent over the network or otherwise transferred to another process. FastormDB, on the other hand, is integrated into the application which eliminates this overhead.

As the application and the database are essentially the same piece of software, the database can be more easily tailored to the application's needs and performance-critical computation can be pushed closer to the data. However, one issue with integrated approach is scalability.

As FastormDB is tightly integrated with application code and it was not originally designed for distributed setup, it is non-trivial to evolve it into that direction afterward. Also, as discussed in section 2.2, distributed databases, in general, have many challenges related to coordination and consistency, especially if multiple nodes are allowed to change database state. Hence the case company opted for a relatively simple model where the database is fully replicated and only one node at a time is allowed to change the database. An overview of FastormDB distributed architecture is presented in section 4.3.

4.2 Transactions in FastormDB

FastormDB features an incremental storage format. Instead of changing existing data, committing creates a snapshot of the new state of the database. Consequently, each snapshot represents a valid database state. Physically a snapshot is a file which contains values changed from the previous snapshot. Snapshots are labeled by incremental timestamps and the most recent snapshot is identified by the *current snapshot timestamp*.

Transactions in FastormDB are always based on some snapshot. Queries in a transaction only see data that exists in the snapshot that the transaction is based on. As snapshots are immutable this achieves transaction isolation. Changes (inserts, updates, and deletes) in a transaction are recorded to secondary data structures. When the transaction commits, it is first rebased on the most recent snapshot to avoid discarding other transactions' work. Changes from secondary data structures are then merged to the latest snapshot, creating a new snapshot.

4.3 FastormDB distributed architecture

A distributed FastormDB deployment consists of multiple compute nodes. A compute node is an instance of RELEX software which includes both application and database code and the database itself. Nodes run on separate machines. Multiple nodes form a cluster and it is assumed that a cluster is deployed to a single data center.

Nodes of a single cluster run the same software but they have different

roles. Examples of roles are transactor, scheduler and worker. *Transactor* is the only node that can make changes to the database, *scheduler* is responsible for starting scheduled batch jobs and *worker* runs jobs started by the scheduler. Roles have their associated state models. For example, strictly speaking there can be multiple nodes with transactor role but only one of them can be primary, or acting, transactor and the rest are in reserve in case the primary transactor fails. However, for the rest of this thesis *transactor* refers to the *primary transactor*.

Structure of distributed FastormDB architecture is presented in figure 4.1. Some parts, such as outbound connections to the user's client, are omitted for simplicity. Linkerd [31] is used for creating a service mesh. All communication in and out of compute nodes goes through local Linkerd client. The only exception are connections to Apache ZooKeeper [9] where cluster-level metadata is stored. State management, e.g. deciding which node is the primary transactor, is handled by Apache Helix [7] which also uses ZooKeeper for persistence. Finally, there is external object storage called OpenIO [40]. Although nodes mainly interact with their local copy of the database, a canonical copy is maintained in OpenIO. This arrangement guarantees the durability of the data even in case of failing nodes.

Distributed FastormDB offers eventual consistency of data. Transactor does not actively synchronize changes with other nodes. Instead, nodes monitor changes of current snapshot timestamp in ZooKeeper and pull snapshots from object storage when a change is detected. In fact, transactor does not wait until the new snapshot has been uploaded to the storage service before changing its local copy of the database. Although this allows the loss of committed data in case the transactor fails before the snapshot upload is complete, it is considered an acceptable tradeoff between consistency and performance and simplicity of the system.

4.4 Commit protocol

As explained in previous sections, in the chosen distributed architecture only the transactor node is allowed to commit transactions. Although this model is simpler than allowing any node to commit, the downside is that worker nodes have to ship their changes to the transactor for committing. A sequence called *commit protocol* is used to do these remote commits. Communication in the commit protocol is done using gRPC which is described in section 4.5. Sequence diagram of the commit protocol is presented in figure 4.2.

Worker initiates commit protocol by sending a request to the transactor. The request contains metadata such as the transaction identifier. When the

Figure 4.1: FastormDB distributed architecture

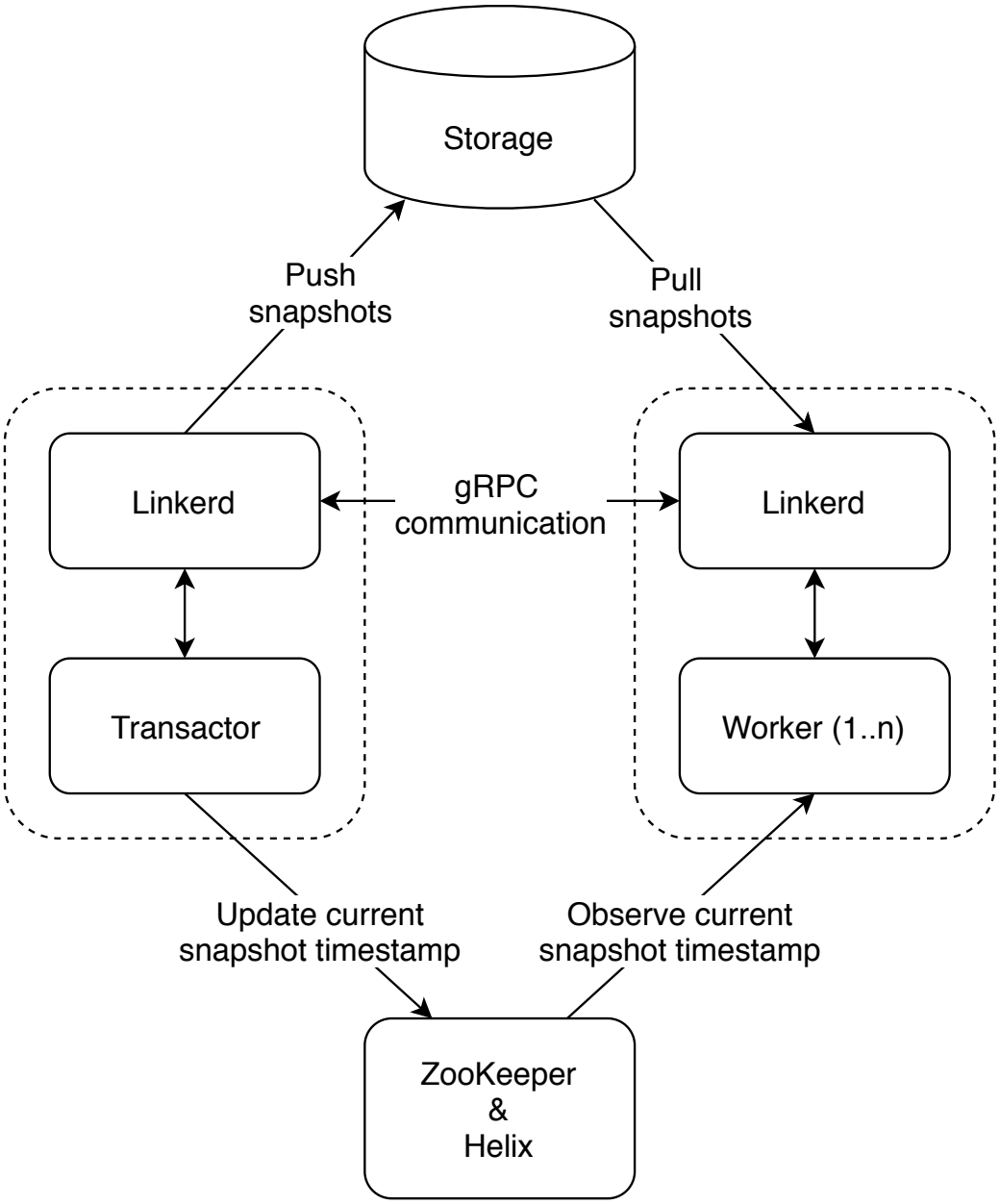
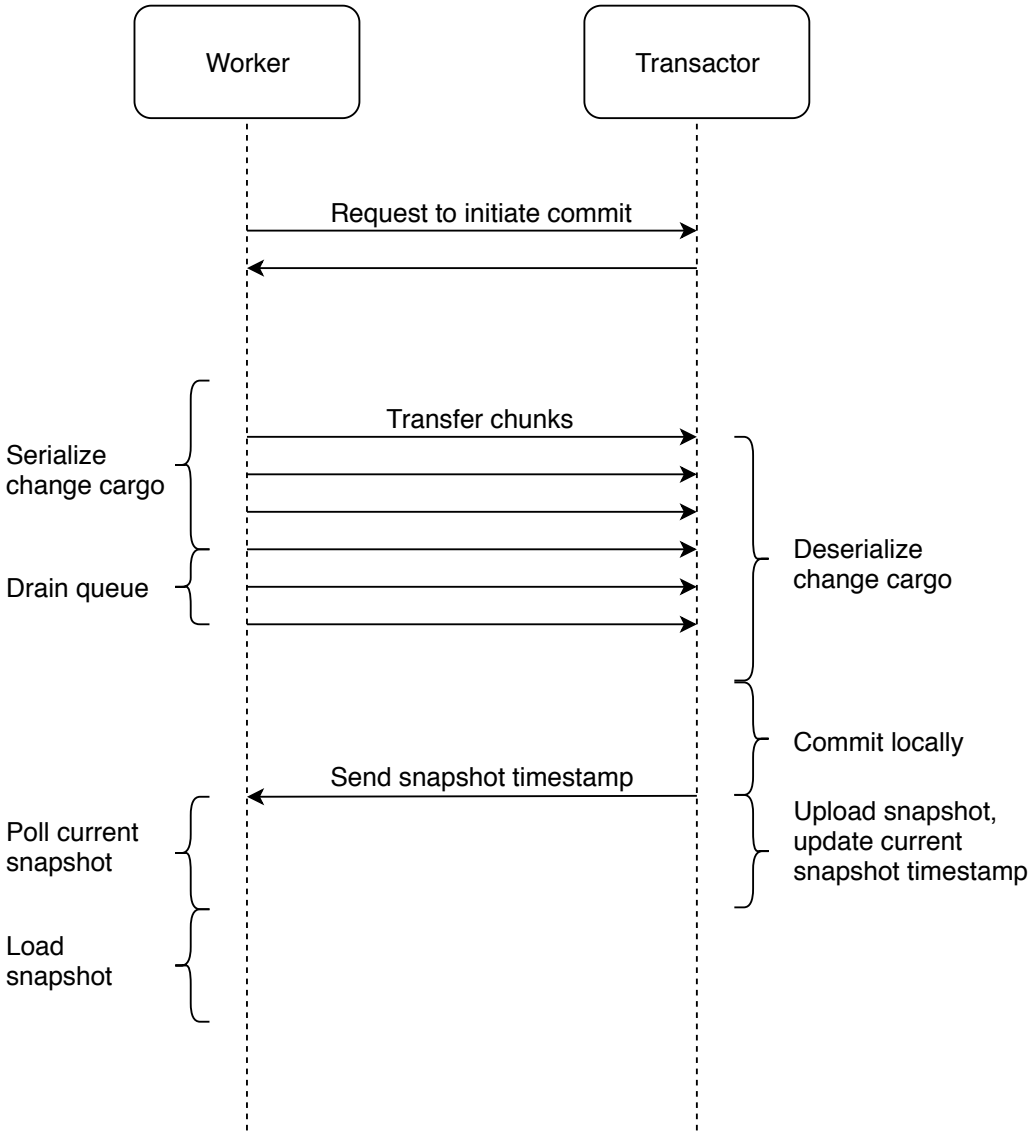


Figure 4.2: Sequence diagram of commit protocol. Time increases from top to bottom. The lengths of phases are not to scale. Arrows indicate flow of gRPC messages.



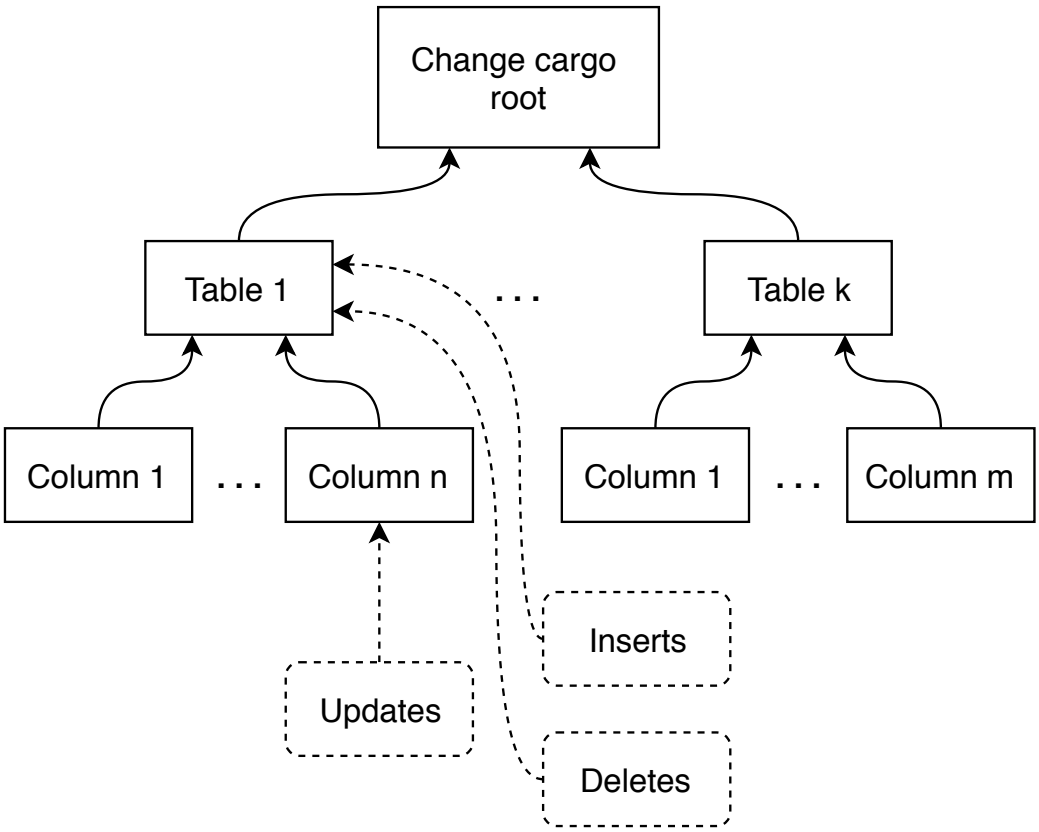
transactor has acknowledged the request, the worker moves into serialization and transfer phase. First, the transaction object is transformed into an object called change cargo. Change cargo contains only essential transaction data and, for example, caches are excluded. Serialization is also easier to implement for change cargo than the full transaction object. The conversion process is fast.

Figure 4.3 describes the structure of change cargo. The root object contains a collection of table objects, one for each changed table. Table objects contain new rows inserted to that table and also deleted rows. Inserts are stored as two-dimensional arrays of primitive data types and deletes are stored as bit sets. Tables also have a collection of column objects which contain updated values of that column. Updates are stored as two-dimensional, primitive arrays similar to inserts. Other than change data, table and column objects also contain small amounts of metadata such as table and column names. Change cargo is structured so that changes to one column have fixed overhead per column (namely the table and column objects). In other words, when more changes are added to the same columns, increasingly larger share of the whole size of the object graph is contained in primitive arrays.

When change cargo has been constructed, the worker starts to serialize it in a new thread. The resulting byte stream can be also compressed. When enough data has been produced, a new chunk is created and added to the transfer queue. Size of each chunk is 63 kB. While serializer thread produces chunks into the queue, another thread drains it and sends the chunks to the transactor. Similarly, on the transactor's end chunks are received into a queue from which they are decompressed and deserialized back into change cargo. It is important to note that serialization, data transfer, and deserialization all happen simultaneously.

When the transactor has fully deserialized the change cargo, it commits the changes to its local copy of the database. At this point, the timestamp of the new snapshot is known and the transactor sends it back to the worker. However, as the transactor uploads the snapshot to the storage service asynchronously it might not be yet available for the worker to download. The worker watches current snapshot timestamp in ZooKeeper until it is equal or greater to the timestamp received from the transactor. When this happens the snapshot is known to be available as the transactor updates the timestamp in ZooKeeper only after the upload has finished. Finally, the worker loads the snapshot which concludes the commit protocol.

Figure 4.3: Change cargo structure



4.5 gRPC

Distributed FastormDB uses gRPC for node-to-node communication. It is a cross-language remote procedure call (RPC) framework [23]. gRPC uses Protocol Buffers [50] as its interface definition language (IDL). An IDL is used to describe RPC services, what methods those services have and message types the methods accept and return. Based on service definition, or schema, it is possible to generate code stubs for services and messages.

gRPC uses HTTP/2 for transport and the rest of the technology stack supports it natively which was a major factor in the decision to use gRPC for communication. A REST-based alternative would have been another option though gRPC has other major advantages. Firstly, gRPC is strongly typed due to the use of proper IDL. Secondly, gRPC supports streaming which is useful when transferring large amounts of data which is the case in the commit protocol. Finally, gRPC uses efficient binary wire format. Specification of gRPC wire format includes an option for message compression. However, most gRPC implementations omit support for that feature which is also the case for the implementation used in this system.

Chapter 5

Evaluated Technologies

An overview of object serialization was presented in chapter 3 and in chapter 4 we presented the system that serves as the platform for the experiments. In this chapter, we discuss and describe the serialization and compression libraries that will be evaluated.

5.1 Serialization

As explained in section 3.1.2, serialization formats can be divided into binary and text-based formats. As there is strong evidence that text-based formats generally perform worse than binary formats [53], only binary formats are investigated. Another important criterion is that the format should not require schema. The object graph that needs to be serialized contains so large variety of complex objects that writing and maintaining schemas would be cumbersome.

5.1.1 Java native serialization

Java offers built-in support for serialization which is often a natural first choice if serialization is needed. It is specified in Java Object Serialization Specification [41]. Java serialization is already used in the considered system so it serves as a good baseline for other serializers.

For class to be serializable, it must implement the `Serializable` interface. It is so-called marker interface meaning it does not actually define any methods. Serialization can be customized by implementing `writeObject` and `writeReplace` methods and the corresponding `readObject` and `readReplace` methods which are used in deserialization. Serialization process checks existence of these methods using reflection. Fields can be also excluded from

serialization using the `transient` modifier.

Actual serialization is implemented by `ObjectOutputStream`. When given an object, its fields are retrieved using reflection and serialized to the underlying `OutputStream`. Regular objects are serialized recursively. Serialization responsibility is delegated to `writeObject` and `writeReplace` methods if either of them is present. `ObjectOutputStream` maintains collection of written objects and on subsequent serializations of the same object only a reference is written. Deserialization is implemented by `ObjectInputStream` which reads bytes from given `InputStream` and reconstructs the object according to process mirroring the one used by `ObjectOutputStream`.

Java serialization is easy to get started with as in many cases it is enough to implement the `Serializable` interface. The downside of this approach, however, is that classes cannot be easily made serializable without modifying their source code, even if the class would not require special handling. This might be an issue with third-party libraries, for example.

5.1.2 Kryo

Kryo is a binary serialization framework for Java which aims for high speed and small size [29]. It used in many applications, such as Apache Spark [13]. Although Kryo is also a Java-specific framework, its format is not compatible with standard Java serialization and the format or serialization process are not well specified.

Kryo does not use marker interface, instead, serialization responsibility is delegated to implementations of `com.esotericsoftware.kryo.Serializer`. Kryo offers several generic `Serializer`s for a variety of use cases. For example, `FieldSerializer` uses Java class files as schema so it has very little overhead but it is also very sensitive to changes in classes. It is suitable for cases where serialized data is transient e.g. sent over the network and discarded after deserialization. The other end of the spectrum is `CompatibleFieldSerializer` which aims for maximum compatibility at the cost of performance. Users can also easily implement new `Serializer`s.

By default, the user has to register all classes that are (de)serialized. This allows Kryo to only write a registration ID instead of full class name during serialization which results in a smaller size. Registration also improves security as unregistered classes cannot be deserialized. Registration also allows the user to specify if a class should be serialized using non-default `Serializer`.

Besides the ability to define new `Serializer`s, Kryo has several configuration options. For the purposes of this study, the two most important are variable-length encoding and usage of unsafe buffers. The first one, variable-

length encoding, instructs Kryo to write integer types (i.e. `int`s and `long`s) using the smallest number of bytes possible. For example, the size of `int` is four bytes in memory but value `1` can be represented with a single byte when using variable-length encoding. Variable-length encoding is more CPU intensive but it can lead to considerable space savings. Variable-length encoding is enabled by default.

Unsafe buffers use `sun.misc.Unsafe` for improved performance. `Unsafe` is an undocumented application programming interface (API) which gives low-level access to Java Virtual Machine (JVM). For example, direct memory access is possible using `Unsafe`. Especially primitive arrays benefit from `Unsafe` as their serialization becomes simple direct memory copy operation (assuming variable-length encoding is disabled). The downside of using unsafe buffers is that serialization format becomes hardware-dependent as endianness is not standardized. For purposes of this study that limitation is acceptable. Moreover, `Unsafe` is not guaranteed to be available on all implementations of the JVM and there is an ongoing plan to remove `Unsafe` altogether [26].

5.1.3 MessagePack

MessagePack is a cross-platform binary serialization format [36]. It specifies types, such as integers and arrays, and format families which specify how types are represented on byte level [37]. MessagePack has no opinion on the use of schema so applications have to agree on how to interpret data beyond formats.

Formats allow values of the same type to be represented efficiently. For example, small positive integers are stored in one byte and large negative values in up to nine bytes. This is somewhat similar to Kryo's variable-length encoding but the exact mechanism is different.

MessagePack implementation for Java can be used directly or via jackson-databind API [25]. Jackson-databind is a library which allows serialization and deserialization of arbitrary Java objects using pluggable formats. It is similar to Kryo in the sense that jackson-databind simply facilitates serialization and actual object-to-bytes conversion is delegated to different serializers (or generators in jackson-databind parlance). The initial plan was to evaluate MessagePack with jackson-databind, however, preliminary testing revealed two major issues. Firstly, jackson-databind adds considerable overhead to serialization. This can be mostly blamed on the general-purpose nature of databind as it was originally designed to be used only with JSON which is a very different type of format. Secondly, MessagePack generator cannot serialize object graphs larger than about 2 GB. For these reasons,

jackson-databind was not considered further.

MessagePack implementation for Java, called msgpack-java, is relatively low-level [39]. It offers methods for writing types in the correct format but it does not offer further support for serialization. In other words, the user is responsible for traversing object graph, deciding which fields to serialize and calling appropriate methods to write the values. This makes msgpack-java rather cumbersome to use but also completely avoids reflection overhead.

5.2 Compression

Compression methods are selected so that they would range from light to strong compression. An important criterion is existing support for Java, either as bindings to native implementation or pure Java implementation. Native implementation is preferred as it is typically faster. Finally, compressors must be able to compress streams as the amount of serialized data is not known beforehand.

5.2.1 Deflate

Deflate is a compression format which uses LZ77 and Huffman coding to compress data [15]. Deflate is very widely used: file formats such as ZIP, GZIP and PNG use Deflate for compression [16, 48]. Java's standard library also has support for Deflate in `java.util.zip` package which was also the implementation used in this study. Due to its widespread use, Deflate serves as a good baseline for compression methods.

5.2.2 Zstandard

Zstandard is "a real-time compression algorithm, providing high compression ratios" [61]. Zstandard has a wide range of compression levels, 1 to 22, however decompression speed is largely unaffected by compression level. For small files, Zstandard offers "training mode" where a separate dictionary is used for both compression and decompression. Dictionaries were not used in this study.

There is no pure Java implementation of Zstandard. However, there are Java bindings, called zstd-jni, to the reference C language implementation [62]. The reference implementation supports multi-threaded compression and decompression. However, zstd-jni 1.3.8-1, which was used for this study, does not support that option.

5.2.3 LZ4

LZ4 is a compression algorithm aiming for extremely fast compression and decompression speed at the cost of compression ratio [33]. Due to its performance, LZ4 is suitable for many real-time applications. Apache Spark, for example, uses LZ4 to compress data transferred between nodes [13].

Unlike many other compression algorithms, LZ4 does not have explicit compression levels. However, there is a high compression variant called LZ4HC. According to the benchmark by LZ4 organization, HC variant increases compression ratio by about 30 % but decreases compression speed by a factor of 20. Decompression performance is the same for both variants. LZ4 features similar support for use of external dictionaries as Zstandard. Neither HC variant or dictionaries were tested in this study.

There is Java support for LZ4 in form of lz4-java package [34]. It offers both pure Java implementation of LZ4 and bindings to reference C language implementation. Neither the Java implementation nor the reference implementation support multi-threaded operation.

5.2.4 XZ / LZMA2

XZ is a file format which uses primarily LZMA2 algorithm to compress data. LZMA and its improved version LZMA2 are strong compression algorithms which were first used in 7z archive format [1]. They are based on the same LZ77 algorithm that Deflate uses.

LZMA2, and by extension XZ, supports different compression levels. The levels range from 0 to 9 with higher values offering stronger compression. Although the default compression level is 6, preliminary testing showed it to be far too slow for the use case evaluated in this study. Surprisingly, higher compression levels also produced larger streams.

XZ Utils, which is software for working with XZ files, supports multi-threaded compression. However, the Java implementation of XZ supports only single-threaded operation although multi-threaded operation is planned [59]. It should be also noted that XZ for Java is pure Java implementation and no bindings for native implementation are available. This might cause performance loss compared to other tested compressors which use native implementation.

Chapter 6

Benchmarks

The system where the evaluation is done was described in chapter 4 and evaluated technologies were presented in chapter 5. In this chapter, we describe what are the actual properties that will be measured, how they will be measured, and what are the benchmark setups.

6.1 Benchmarking methodology

Especially in managed runtime systems, such as the Java platform, there are many factors affecting performance, some of them non-deterministic [20]. This has led to the development of many Java benchmark suites, newer iterations trying to fix deficiencies found in the earlier ones [11, 49, 54]. Although these suites are more aimed at the platform developers, many of the dynamic effects of Java platform, such as garbage collection and just-in-time (JIT) compilation, have to be taken into account in application benchmarking as well.

There are different methodologies for handling the dynamic factors. For JIT compilation, it is common to wait that the application reaches a steady state. That is, the operation to be measured is first run one or more times as warmup before the actual measurements are done [20]. This also prevents class loading from affecting the measurements. Garbage collection is more difficult to account for as it is usually under the discretion of the JVM when to run it. One approach is to force garbage collection before each iteration which keeps runtime of different iterations comparable [20]. On the other hand, garbage collection is a price that must be paid in real use and ignoring it might provide misleading results if the measured operation produces a lot of garbage. In this thesis, both warmup iterations and forced garbage collection are used.

6.2 Measurements

When benchmarking complex systems, such as the one presented in this thesis, it can be difficult to isolate the effect of an individual component. One solution is synthetic benchmarks where relevant components are benchmarked in isolation. As synthetic benchmarks are usually easy to run due to requiring little setup, they are useful for testing different configurations and prototyping enhancements. However, care should be taken when interpreting the results of synthetic benchmarks as by definition they exclude the effects of other components that would be present in normal use. Therefore results of synthetic benchmarks should be compared to results of benchmarks done in production-like setups to ensure that valid conclusions can be drawn. In this thesis, we perform both synthetic benchmarks and throughput benchmarks simulating real-world conditions.

6.2.1 Serialization and deserialization time

In the context of this thesis and the described system, a logical synthetic benchmark is measuring (de)serialization and (de)compression in isolation. When compared to results of throughput benchmark (described later), results from these synthetic benchmarks can be used to verify if throughput performance is actually affected by (de)serialization and (de)compression performance. For example, if throughput performance closely follows serializer performance it indicates there are no bottlenecks caused by the network or another component. On the other hand, if throughput performance is the same regardless of serializer then it is likely that the production-like setup contains some bottleneck (assuming that there are differences between serializers in the synthetic benchmarks).

6.2.2 Size of serialized data

The amount of data that different serializers produce is not inherently relevant value for the goals of this thesis. However, measuring size helps with understanding the results of other benchmarks. For example, if it seems that highest throughput is achieved with highest compression ratios, even at the cost of raw serialization speed, it might indicate that the process is bottlenecked by bandwidth or other issues in node-to-node communication. Size of serialized data can be measured as a byproduct of other benchmarks.

6.2.3 Throughput time

As has been mentioned earlier, the most interesting property is the overall throughput time. There are several reasons for this. Most importantly, the time it takes to commit directly adds to the overall runtime of a batch job since the results of the job are not available for the rest of the system before they have been committed. Timely availability of the results is important for many use cases, especially if the changes were done due to users' actions instead of a scheduled batch job.

The commit protocol, described in 4.4, can be divided into logical phases. All of these phases, like applying the changes to local database copy, are not affected by the choice of serializer and compressor. Therefore when measuring throughput time, it is useful to also measure how long individual phases take.

The most important phase is the serialization phase where the change cargo is serialized and possibly compressed. The resulting data is split into chunks which are put into send queue. During the serialization phase chunks are also being sent to the transactor. Another notable phase is *queue draining* which immediately follows the serialization phase. If the serialization process produces data much faster than chunks can be sent there might be a considerable amount of chunks queued after serialization. The choice of serializer and compressor affects how long the queue will be and, consequently, how long it takes to drain it.

6.3 Benchmark setups

The basic software and hardware configuration and system setup are shared between all performed benchmarks. Next, we describe both the shared configuration and individual setups of each benchmark.

6.3.1 Software and hardware configuration

Evaluated serializers are listed in table 6.1. Java serialization and MessagePack have no configurable behavior. For Kryo, all serialized classes are registered. For most classes, the default `FieldSerializer` is used. For classes which require special handling (e.g. they implement custom `writeObject` method), a serializer based on `FieldSerializer` which implements the required behavior is used. In addition to these common settings, two different Kryo configurations are evaluated. *Kryo (default)* is equivalent to default Kryo configuration which uses variable-length encoding and regular buffers. *Kryo (unsafe)* is otherwise similar but it uses unsafe buffers with variable length encoding disabled.

Serializer	Implementation	Version
Java	JDK8	Oracle 1.8.0_162
Kryo	com.esotericsoftware.kryo	5.0.0-RC2
MessagePack	org.msgpack.msgpack-core	0.8.16

Table 6.1: Evaluated serializers

Evaluated compressors are listed in table 6.2. All serializers are tested with all compressors and also with no compression. Deflate and Zstandard are tested with their default compression levels. XZ is tested with the fastest compression level.

Compressor	Implementation	Version	Compression level
Deflate	JDK8	Oracle 1.8.0_162	6 (default)
Zstandard	com.github.luben.zstd-jni	1.3.8-1	3 (default)
LZ4	org.lz4.lz4-java	1.5.0	-
XZ	org.tukaani.xz	1.8	0

Table 6.2: Evaluated compressors

Synthetic benchmarks are run on a server with four Intel Xeon E5-4640 processors running at 2.40 GHz and with 768 GB of memory. Throughput benchmarks are run on two similar servers that are connected by a 1 Gbit/s link. All benchmarks are run on Oracle HotSpot JVM 1.8.0_162. Maximum heap size in all benchmarks is set to 400 GB.

6.3.2 Data generation

The amount of work done by serializer and compressor (similarly for deserializer and decompressor) depends on the amount and characteristics of input data. Therefore, the input data and the method for its generation are relevant parts of the performed benchmarks. All benchmarks use the same method which is described in the following.

The input data for serializer is the change cargo object described in section 4.4. Change cargo contains all changes done in a transaction. This means that generating input data is essentially the same as generating a certain amount of changes in a transaction. These changes can be new rows, updates to existing rows and row deletions. In typical use of the system, large commits

Target size	Actual size	Changes to larger table	Changes to smaller table
100 MB	100.7 MB	2 700 000	27 000
1 GB	1005 MB	27 000 000	270 000
10 GB	10 040 MB	270 000 000	2 700 000
100 GB	100 500 MB	2 700 000 000	2 700 000

Table 6.3: Input sizes for benchmarks. The number of changes is divided equally between inserts and updates.

contain few deletions so for simplicity only inserts and updates are generated. Deletions also take a relatively small amount of memory to store. Changes are generated to two tables with the majority of the changes done to the same, larger table. This kind of distribution is also typical for large commits.

All benchmarks are run with different input sizes: 100 MB, 1 GB, 10 GB, and 100 GB. These sizes describe the number of bytes required to store the raw change data in memory. The number of bytes required to store pointers, object headers, and similar is not included in the size. Because the input is a complex object graph it is non-trivial to generate change cargo with the exact amount of raw data. Therefore the actual inputs are slightly larger than the target sizes. The deviation is small, less than 1 % in all of the cases. However, this does not affect the results as all benchmarks use the same input. Input sizes are listed in table 6.3.

6.3.3 Serialization benchmark

To measure serialization and compression performance, a change cargo is first created using the method described in section 6.3.2. This change cargo is then serialized and compressed with each serializer-compressor combination as a warmup. Warmup phase is required to mitigate the effects of JVM's dynamic behavior such as JIT compilation.

During the warmup, the size of the resulting data is also measured. After the warmup, each combination is used to serialize and compress the same change cargo five times and time for each iteration is measured. Before each iteration, JVM is also instructed to run garbage collection. This benchmark is repeated for all change cargo target sizes.

6.3.4 Deserialization benchmark

To benchmark deserialization and decompression, the generated change cargo is first serialized and compressed with the corresponding serializer-compressor combination. This data is written to array in memory so that serialization will not create a bottleneck for deserialization performance. Similarly to serialization benchmark, deserialization benchmark starts with a warmup phase where each deserializer-decompressor combination is used to deserialize the data once. After the warmup, each combination is used five times to deserialize and decompress the data and the runtime of each iteration is measured. Garbage collection is run before each iteration. Deserialization benchmark is repeated for all change cargo target sizes.

6.3.5 Throughput benchmark

The throughput benchmark is run on a two-node cluster. The nodes run on separate but hardware-wise identical servers described in section 6.3.1. One node is a worker and the other has all remaining roles, including the transactor role. Throughput benchmark includes similar warmup phase as other benchmarks. Garbage collection is also run before each iteration.

The worker first generates changes to a transaction as described in section 6.3.2. The transaction is then committed according to the protocol described in section 4.4. That is, the worker serializes and possibly compresses the change cargo and transfers it to the transactor where the cargo is deserialized and applied to the database. The time is measured from the start of the commit (i.e. data generation time is not measured) to the point where the worker has loaded the snapshot from the storage service. Times for different phases of the commit is measured as well. Five iterations are done per serializer-compressor combination. The benchmark is repeated only for 100 MB, 1 GB, and 10 GB change cargo target sizes.

Chapter 7

Results

Benchmarks were presented in chapter 6. Results of those benchmarks are discussed in this chapter. Detailed values are available in appendices A to D.

7.1 Serialization benchmark

Results for serialization benchmark are tabularized in appendix A. The tables list average runtime for each serializer-compressor combination and also relative score and absolute runtime difference compared to Java serializer without compression. The score is calculated by dividing the average runtime of the combination by the average runtime of the baseline. Therefore a lower score is better. Average runtimes are also presented in figure 7.1.

When comparing serializers without compression, Kryo (unsafe) is the fastest for all input sizes. The difference is clearest in the 100 MB and 1 GB benchmarks where its runtime is only third of Java's. The relative performance of other serializers is more varied. In the 100 MB benchmark, Kryo (default) is only 6 % slower than Java and MessagePack is 40 % faster. On the other hand, in the 1 GB and 10 GB benchmarks, Kryo (default) is 45 % and 47 % slower than Java, respectively, and MessagePack is 13 % and 5 % slower than Java. In 100 GB benchmark the difference is even larger: Kryo (default) is 67 % and MessagePack is 46 % slower than Java.

In general, the use of compression makes runtime considerably higher. Even in the best case, the runtime is almost three times higher compared to no compression. XZ is the slowest to compress with Deflate being only 10 % to 30 % faster. Both XZ and Deflate become increasingly slower as the input size grows. For example, in 100 MB benchmark, Java serializer's runtime is 32 and 36 times higher when compressed with Deflate and XZ,

respectively, compared to no compression. In the 100 GB benchmark, these numbers are 51 for Deflate and 65 for XZ. Both Deflate and XZ have very similar performance regardless of the serializer.

For Java and MessagePack, LZ4 is clearly the fastest compressor with scores ranging from 2.7 to 4.0. Zstandard is roughly half as fast with scores ranging from 6.7 to 9.0. Both LZ4 and Zstandard have similar performance with Java and MessagePack for all input sizes.

For Kryo (default) and especially Kryo (unsafe) the results are very different. With Kryo (default), the performance of LZ4 and Zstandard is almost identical: less than 4 % difference for all input sizes. With Kryo (unsafe), LZ4 is up to 45 % slower than Zstandard but their performance is still much more similar than with Java and MessagePack. For all input sizes, LZ4 increases runtime of both configurations of Kryo three to four times more compared to Java and MessagePack. In fact, Kryo (unsafe), which is always the fastest when used without compression, is the slowest for all input sizes when LZ4 compression is used.

Performance of serializer-compressor combinations as function of input size in serialization benchmark is presented in figure 7.2. As can be seen from the graph, for most combinations the runtime increase is sublinear in size of the input data. A major exception is Kryo (unsafe) without compression where increase from 1 GB to 10 GB is superlinear.

7.2 Deserialization benchmark

Results for deserialization benchmark are tabularized in appendix B. Similarly to serialization benchmark results, the tables list average runtime for each serializer-compressor combination and also relative score and absolute runtime difference compared to the reference serializer. Average runtimes are also presented in figure 7.3.

Kryo (unsafe) is the fastest deserializer for all input sizes when the input is not compressed. The difference is again clearest for 100 MB and 1 GB inputs where Kryo (unsafe) is 58 % and 53 % faster compared to Java. In the larger, 10 GB and 100 GB cases, the difference is still 17 % and 23 %, respectively. In the smallest, 100 MB case, MessagePack is 37 % and Kryo (default) is 6 % faster than Java. However, with larger sizes Java becomes second fastest: MessagePack is 0.7 % to 12 % slower and Kryo (default) 11 % to 17 % slower.

Having to first decompress data makes deserialization slower although the effect is smaller than with compression on serialization benchmark. XZ is the slowest also in decompression, with both Kryo configurations and Mes-

Figure 7.1: Serialization runtimes in seconds. Note the logarithmic scale.

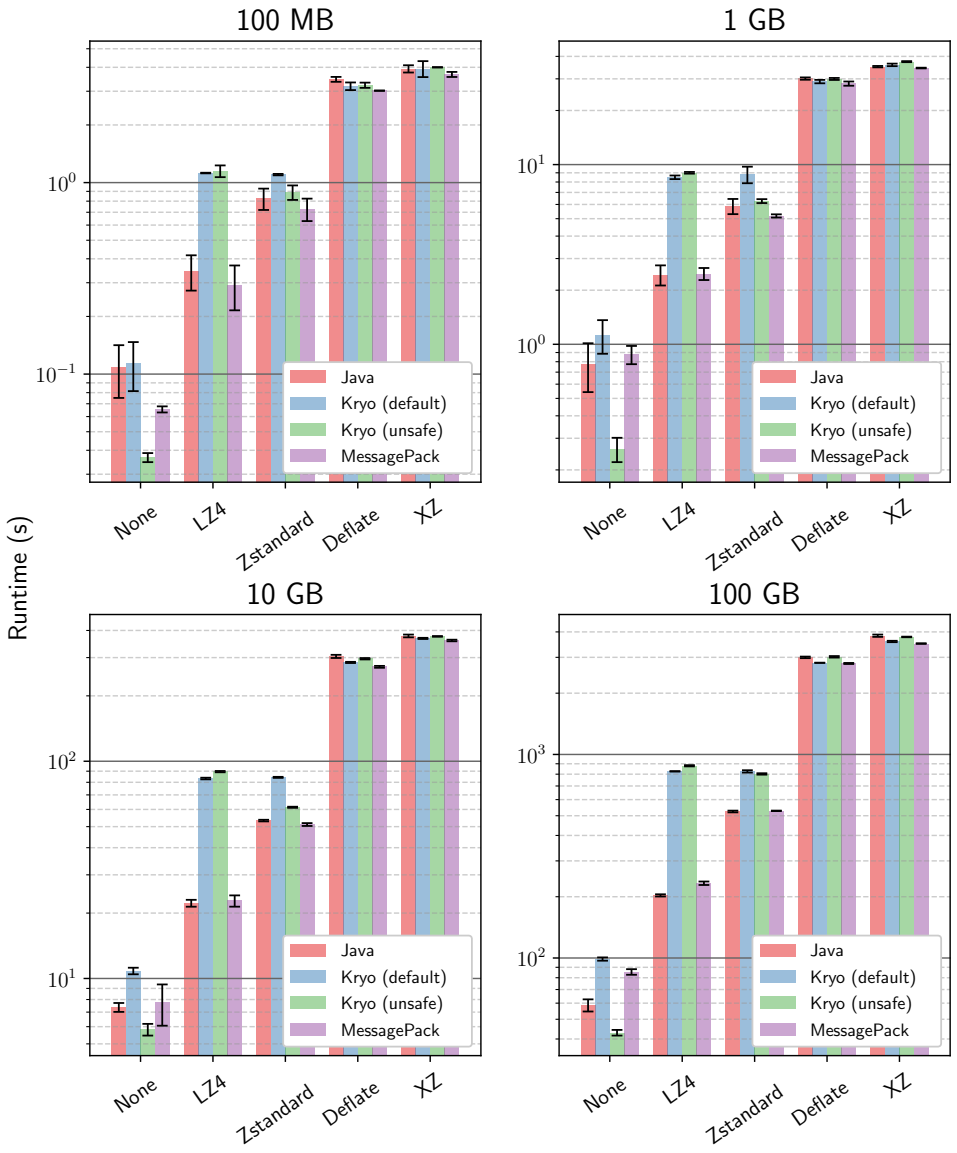
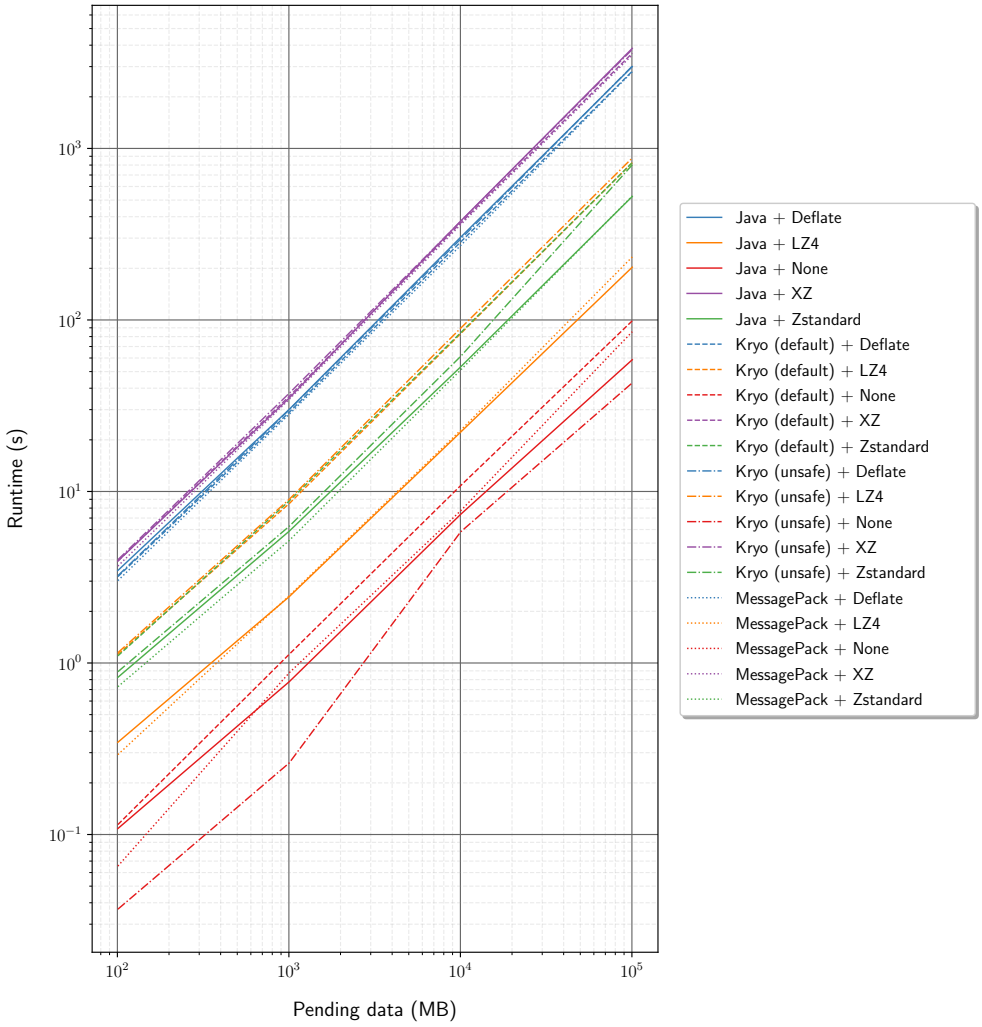


Figure 7.2: Serialization performance as function of input size. Note the logarithmic scale on both axes.



sagePack having scores from 21 to 22 for all input sizes. For Java, however, the scores are only around half of that, from 11 to 14. Deflate is the second slowest to decompress, with scores ranging from 3.7 to 4.1 for the three largest input. Java is an exception, having a score around one point higher. In the 100 MB case the scores are somewhat higher, ranging from 5.1 to 5.8 for others and 8.8 for Java.

LZ4 is the fastest compression method. It is at most 53 % slower than the baseline, Java serializer without compression, and in many cases much faster. Specifically, with Kryo (unsafe) LZ4 is at most 5 % slower and in 100 MB and 1 GB up to 25 % faster than the baseline. It should be also noted that Kryo's slowness with LZ4 in serialization benchmark is not visible in the deserialization benchmark and performance of LZ4 is more consistent overall.

Performance-wise Zstandard is somewhere between LZ4 and Deflate. In the 1 GB, 10 GB, and 100 GB cases its score ranges from 2.5 to 4.3. In the 100 MB case, the score range is similar for Kryo and MessagePack but Java has a score of 12.8 which is three times larger than the second highest.

When looking at absolute runtimes, deserialization is 0 % to 91 % slower than serialization when data has not been compressed. However, with compressed data the situation becomes the opposite and deserialization, including decompression, is 27 % to 92 % faster than serialization and compression.

Performance of serializer-compressor combinations as function of input size in deserialization benchmark can be seen in figure 7.4. Deserialization runtime, in general, is sublinear in the input data size. Increase for Kryo (unsafe) from 1 GB to 10 GB is again superlinear though less than in the serialization benchmark.

7.3 Size of serialized data

The amount of data serializer-compressor combinations produce is tabularized in appendix C. No tested serializer-compressor had any variance in output size for given input so the listed values are exact. The tables list score and absolute difference of each combination compared to the reference combination, Java serializer without compression. The tables also include compression ratio which is commonly defined as

$$\text{Compression ratio} = \frac{\text{Uncompressed size}}{\text{Compressed size}}$$

Compression ratios are calculated separately for each serializer. Serialized data sizes are also presented in figure 7.5.

Figure 7.3: Deserialization runtimes in seconds. Note the logarithmic scale.

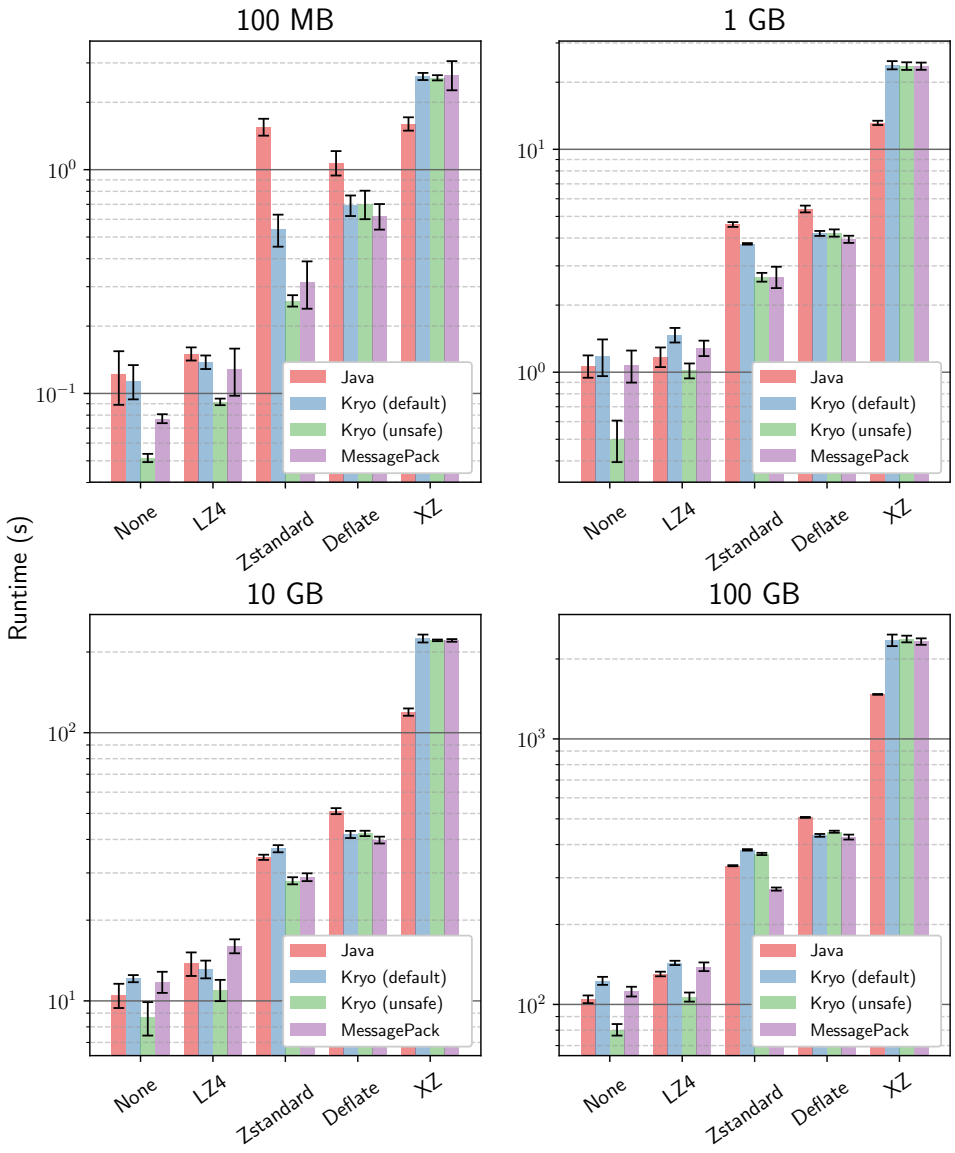
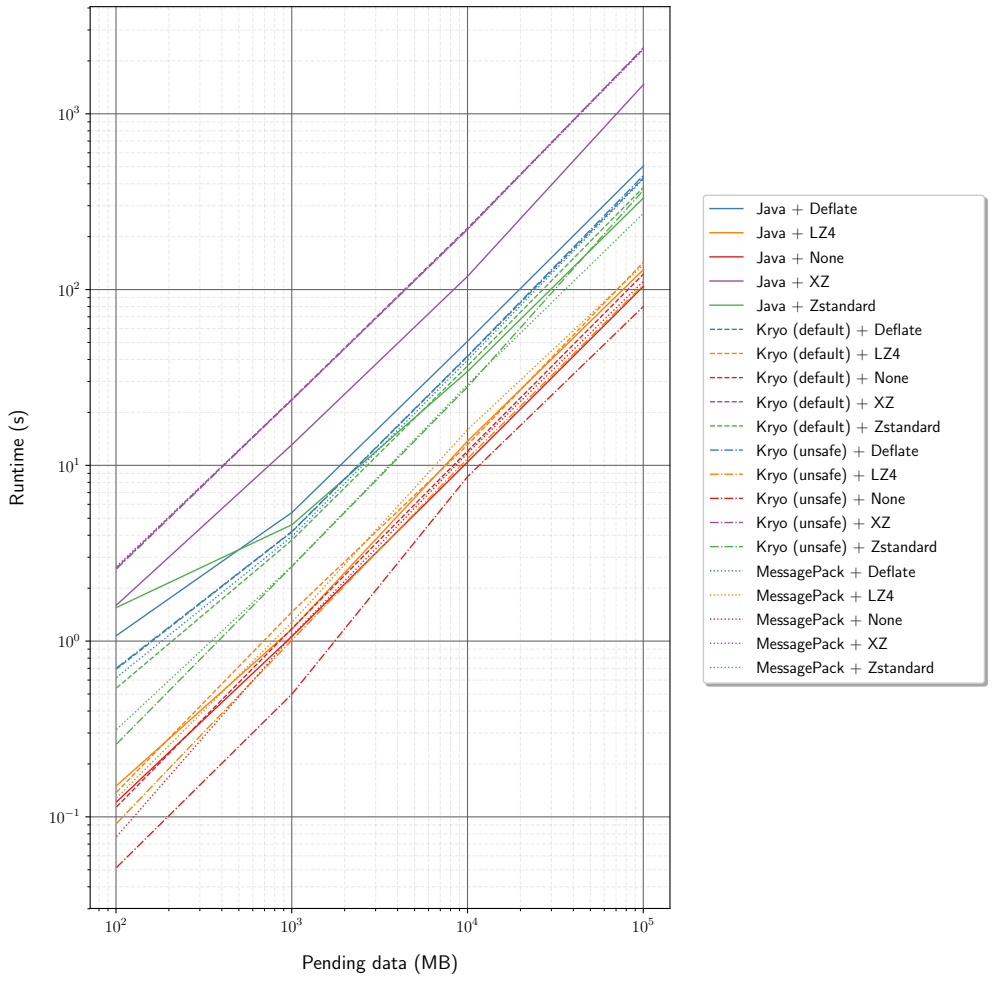


Figure 7.4: Deserialization performance as function of input size. Note the logarithmic scale on both axes.



The size of serialized data produced by Java serializer and Kryo (unsafe) is almost identical: for all input sizes, the difference is less than 1 %. The size of serialized data is only marginally larger than the size of pending data. The overhead is less than 7 % in the 100 MB case and decreases to less than 1 % in the 100 GB case. MessagePack and Kryo (default) also have very similar results with each other. Their sizes are within 1.5 % of each other, with Kryo (default) producing slightly less data. Amount of serialized data produced by MessagePack and Kryo (default) is up to 15 % less compared to the size of pending data.

LZ4 is the lightest compression method. When combined with Java serializer and MessagePack, there is less than 1 % difference in output size. Outputs of both Kryo configurations are compressed to very similar sizes as well. However, with Kryo the final compressed size is up to 14 % larger than with Java serializer or MessagePack. Zstandard compresses data to around half of the size of LZ4. Similarly to LZ4, Zstandard compresses data less effectively when used with either Kryo configuration compared to Java serializer and MessagePack.

Deflate produces similar sizes compared to Zstandard with little variation between different serializers. The size difference between the two compressors is less than 7 % with Zstandard being better with Java serializer and MessagePack and Deflate with Kryo. XZ is the heaviest compressor. Compared to Zstandard and Deflate, using XZ results in up to 40 % less data. There is some difference between serializers, however. XZ compresses the output of Java serializer the most and Kryo (default) the least; the difference between the two is 9 % to 10 %. MessagePack and Kryo (unsafe) fall in the middle.

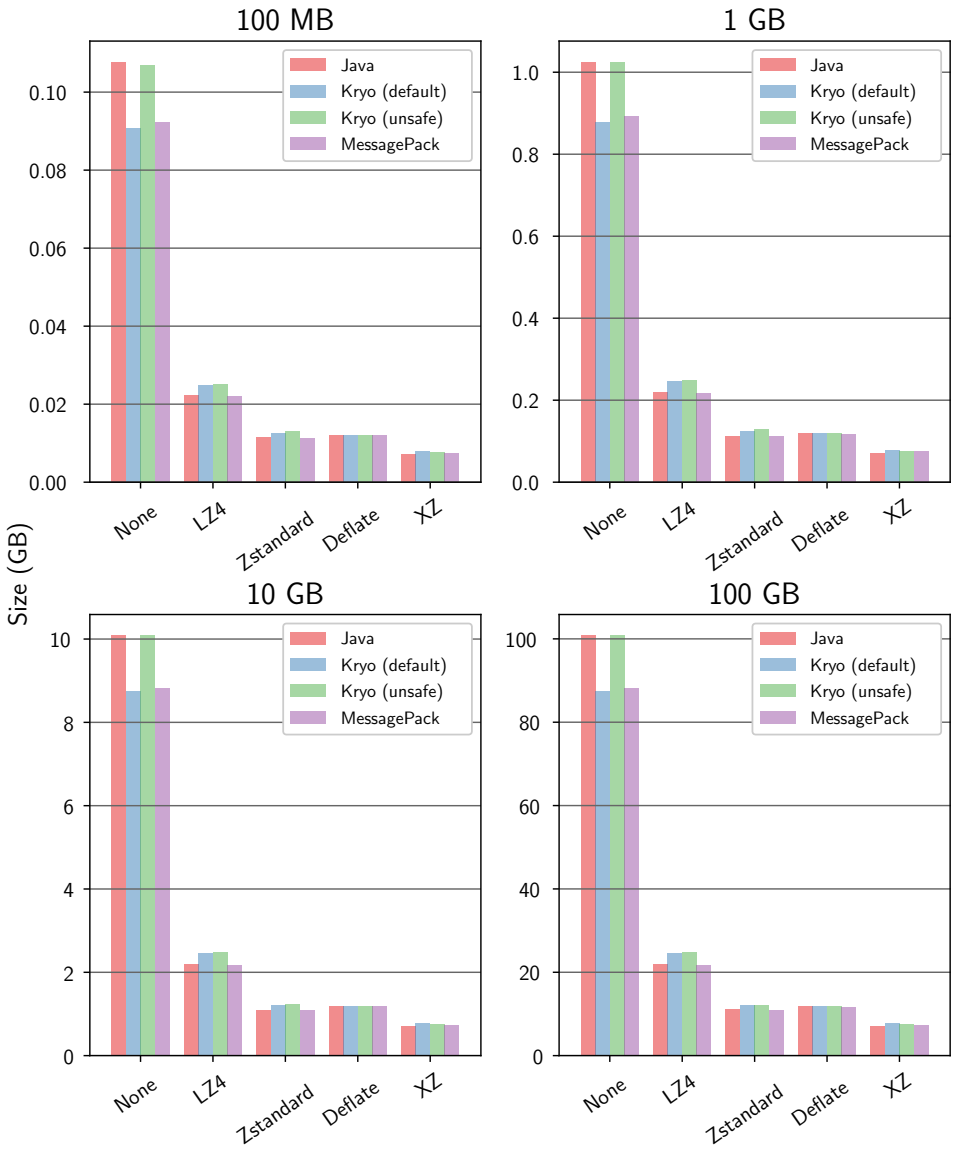
The compression ratio of all compressors is highest with Java serializer. However, it should be noted that the compression ratios are not directly comparable between serializers as inputs to compressors differ. Compression ratios are also generally lower for larger input sizes.

7.4 Throughput benchmark

Results for throughput benchmark are tabularized in appendix D. In tables D.1 to D.3 *average runtime* is the average time it took worker to complete the whole commit and *S+Q* is the time taken to serialize change cargo and send all chunks to the transactor. Score is calculated using average runtime and it represents runtime relative to Java serializer's performance. Average runtimes are also presented in figure 7.6.

In the 100 MB case, Kryo (unsafe) without compression yields the highest throughput, being 9 % faster than the baseline. Uncompressed Kryo (default)

Figure 7.5: Size of serialized data in gigabytes.



and MessagePack also achieve lower runtime than the baseline, although only by 4 % and 2 %, respectively. However, second and third fastest combinations are Java and MessagePack with LZ4 compression, achieving 7 % and 5 % lower runtime than the baseline. LZ4 is slower with Kryo, resulting in 6 % higher runtime with the default configuration and 9 % higher with the unsafe. Using Zstandard for compression with Kryo and MessagePack results in 5 % to 6 % increase in runtime while with Java serializer the runtime is essentially the same. Depending on the serializer, using Deflate results in 35 % to 50 % higher runtime compared to the baseline. For XZ this range is 50 % to 60 %.

In the 1 GB case, Kryo (default) without compression is the fastest although the difference to the baseline is less than 2 %. Kryo (unsafe) is barely faster than Java and MessagePack is 5 % slower. Using any compression method with any serializer increases runtime compared to the baseline. However, LZ4 is rather close with Java serializer and MessagePack: the increase is only 2 % and 3 %, respectively. Notably, MessagePack is still faster with LZ4 than without compression. For Kryo, using LZ4 increases runtime more, 16 % for the default configuration and 17 % for the unsafe. Compressing output of any serializer with Zstandard increases runtime 7 % to 13 %, with Deflate 65 % to 73 % and with XZ 79 % to 88 %.

In the 10 GB case, regardless of the serializer, using any compression method results in higher runtime compared to the situation where no compression is used. Java serializer achieves the best runtime although the difference to Kryo (default), which is the worst serializer, is only 3 %. LZ4 increases runtime with Java and MessagePack by 5 % and 9 %, respectively, but with Kryo (default) and (unsafe) the increase is 24 % and 32 %, respectively. The increase caused by Zstandard is 23 % to 24 % except with MessagePack when the increase is only 18 %. Deflate and XZ are again slowest. Increase in runtime when using Deflate ranges from 104 % to 114 % and with XZ from 127 % to 139 %.

Throughput runtime as function of input size is presented in figure 7.7. Like in serialization and deserialization benchmarks, throughput runtime increases sublinearly in the input size. However, sublinearity is stronger in throughput benchmark and the increase in runtime as the input size grows is more uniform.

Difference of total runtime and time taken to serialize and drain chunk queue stays almost constant for all combinations for given input size. The difference, which mostly represents the time taken to write and upload the snapshot, is 3.5 s to 4.2 s in the 100 MB case, 18 s to 19 s in the 1 GB case and 56 s to 61 s in the 10 GB case. For the baseline combination, the differences are 55 %, 40 % and 19 % of the total runtime in 100 MB, 1 GB and 10 GB cases, respectively.

Figure 7.6: Throughput runtimes in seconds. Note the logarithmic scale.

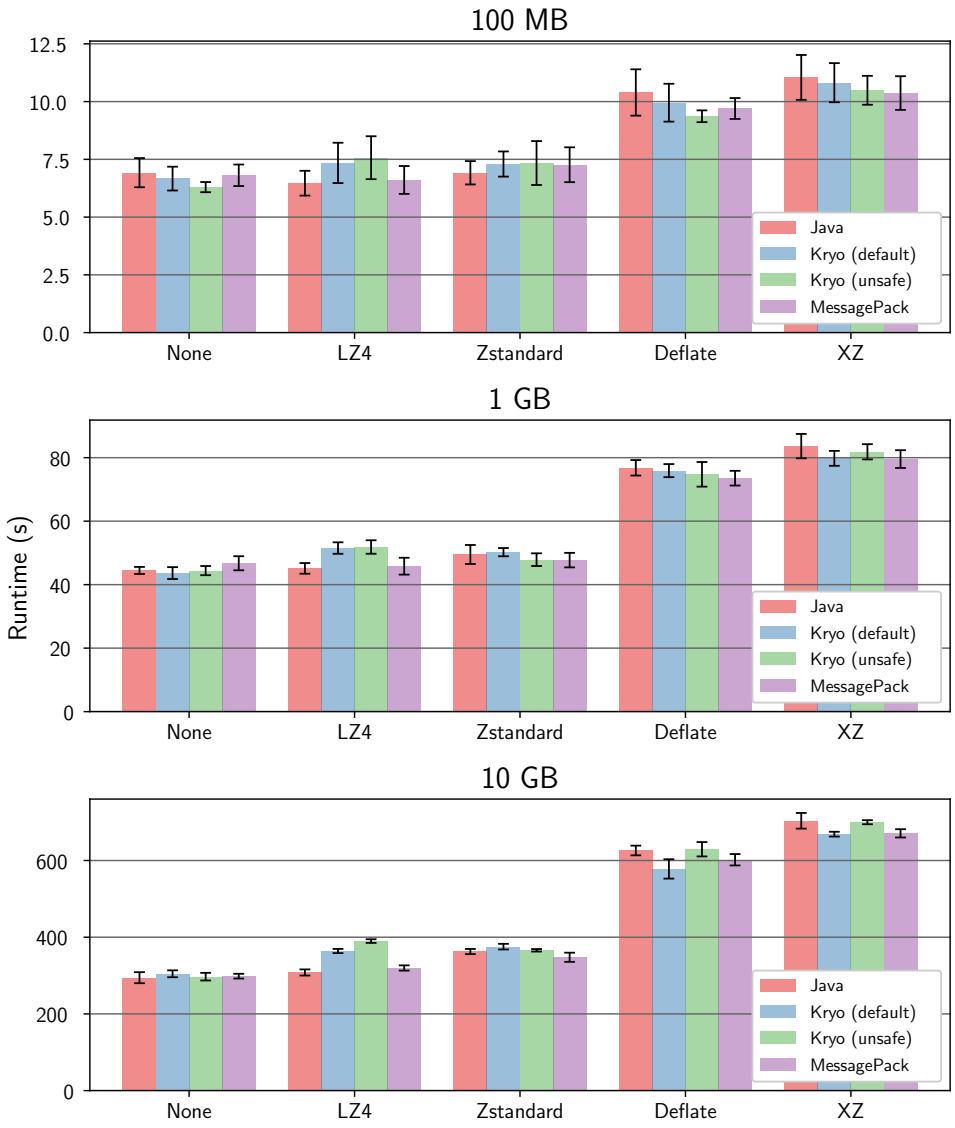
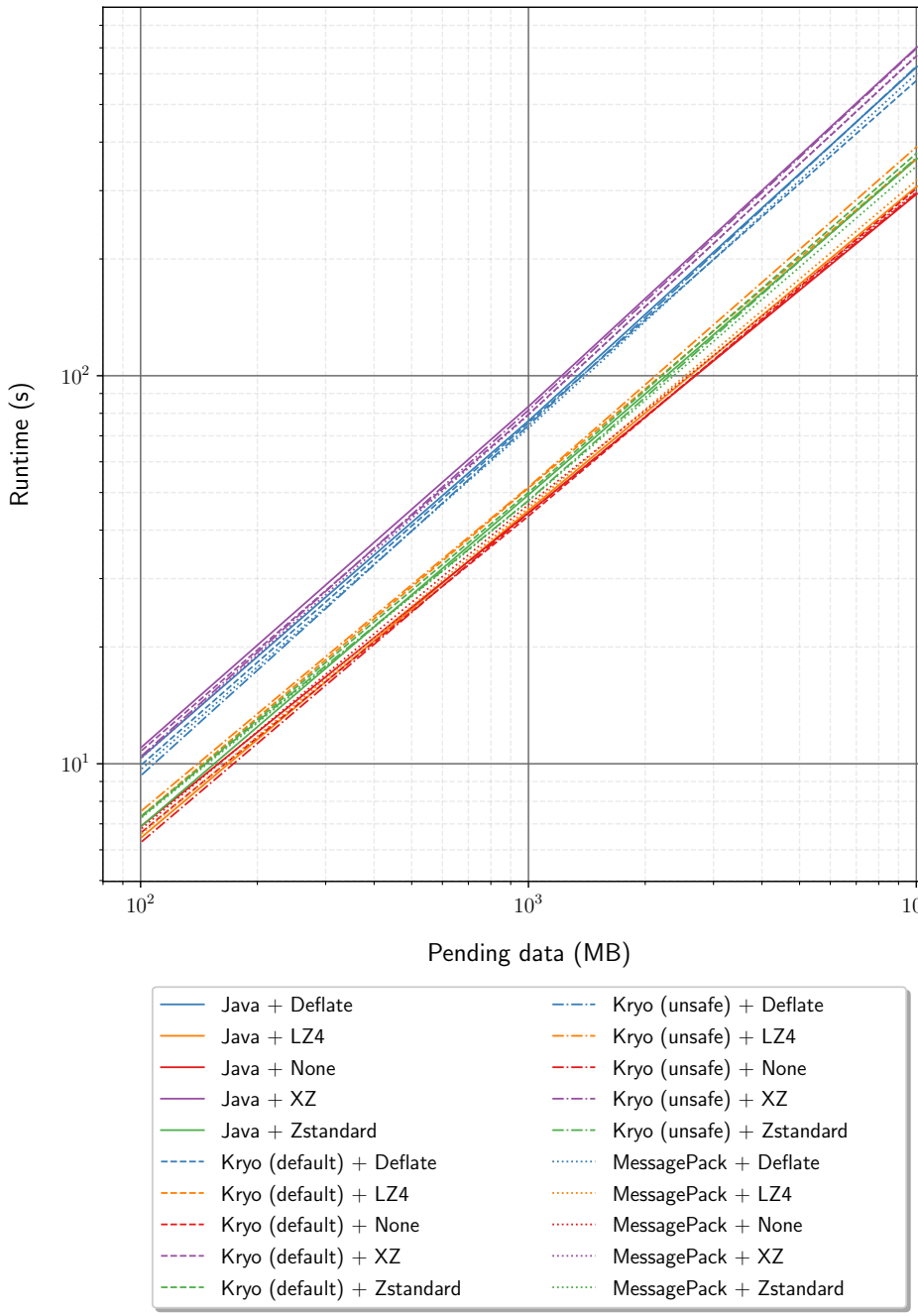


Figure 7.7: Throughput runtime as function of input size. Note the logarithmic scale on both axes.



Chapter 8

Analysis and Future Work

In chapter 6 we described our experiments and in chapter 7 we presented the results. In this chapter, we analyze the results. The chapter is concluded by recommendations about future work.

8.1 Analysis

Performed measurements were presented in section 6.2. The analysis is organized around each aspect described in the aforementioned section. The contrast between practical and synthetic benchmarks and applicability of obtained results is also discussed in section 8.1.4.

8.1.1 Serialization performance

Kryo (unsafe) provides the best serialization performance. This is expected as most of the serialized data is in primitive arrays which can be copied using fast, internal APIs. On the other hand, Kryo (default) is on par with Java only in the 100 MB benchmark. In 1 GB and 10 GB cases it is almost 50 % slower than Java and it peaks at 69 % slower in the 100 GB benchmark. The relative slowness is likely caused by variable-length encoding which Kryo uses by default and which also results in 13 % to 16 % less data than with Java. Although MessagePack is faster than Kryo (default), their performance profiles are similar. MessagePack starts at 40 % faster than Java in 100 MB benchmark, slows to 10 % slower in 1 GB and 10 GB cases and peaks at 46 % slower in 100 GB case. The slowness of MessagePack is offset by output sizes similar to Kryo (default).

In other studies, Kryo has also been found to be faster than Java serializer. Both Sikdar et al. [53] and Petersen et al. [45] measured the runtime of

Kryo to be around 10 % to 50 % of Java serializer's runtime depending on the input. Although the results are similar to what Kryo (unsafe) achieved in this study, the important difference is that the measurements were done with the default configuration of Kryo. In both studies, the inputs were much smaller (from several hundred bytes up to 20 kilobytes) and, more crucially, contained relatively little integer values than the inputs in this study. This shows how large difference variable-length encoding makes.

Adding changes to a change cargo creates data mostly into primitive arrays at the leaves of the object graph. Parts of the object graph have a fixed size which is unrelated to the size of the change cargo. As the change cargo grows, the relative size of the fixed parts decreases and the graph becomes "data-heavy". That is, there are more primitive data and fewer objects. As both Kryo configurations' serialization performance relative to Java serializer becomes worse as the size of the input data grows it would indicate that Kryo is more effective at serializing objects than Java serializer. Java serializer writes more data and does more reflective operations per object which supports this idea.

Based on MessagePack performance, it is rather beneficial if using reflection for traversing the object graph can be avoided. MessagePack would likely be close to Kryo (unsafe) performance or even surpass it if the input was "object-heavy". However, as MessagePack uses variable-length encoding for integers it loses in performance for Java serializer on larger input sizes. On the other hand, Petersen et al. [45] found that MessagePack is over six times slower than Java serializer which rather directly contradicts this conclusion. How they implemented object graph traversal might explain the large difference as they measured MessagePack to be only 50 % slower than Java when used via jackson-databind.

8.1.2 Deserialization performance

Kryo (unsafe) has also the best deserialization performance although variations in relative performance are smaller overall. Except for Kryo (default) and MessagePack in the 100 MB case, deserialization is slower than serialization. It is not immediately clear why this is the case. One possible reason is that as serializers reconstruct the object graph they have to allocate memory for objects whereas in serialization benchmark the target array was pre-allocated. On the other hand, deserialization is much faster when data is compressed. This suggests that the buffer where serialized data is read from is a limiting factor as considerably less data must be read if it is compressed which results in better runtime. If this is the case then heavier compressors get more advantage. Regardless, these results would not seem to affect the

overall ranking of serializers and compressors.

8.1.3 Size of serialized data and compression

Despite Kryo having more efficient serialization format, even in the 100 GB case Kryo (unsafe) produces only 2.5 MB less data than Java. For the tested input sizes this difference is insignificant. Variable-length encoding that Kryo (default) and MessagePack use, however, results in considerably less serialized data. The two serializers use a different type of variable length encoding which is likely the reason why MessagePack's data is one percentage point larger. MessagePack also does not write any metadata per object but as pointed out in the comparison of Java serializer and Kryo (unsafe), this results in insignificant savings at the tested input sizes.

Space optimizations done by serializers are largely irrelevant when using separate compression. For example, Java serializer produces up to 17 % more data than MessagePack but with any compressor, the difference in final size is less than 1 %. It is also interesting to note that although Java serializer's output is the largest of any tested combination, the smallest size for any input is achieved when its output is compressed with XZ. Best compression ratios were also achieved with Java serializer. These results also indicate that the format of Java serializer is rather verbose.

It is rather clear that if compression is required, large amounts of serialized data can and should be compressed with a dedicated compressor. However, optimizing for space at the serialization phase might be useful if the amount of data is small, say in kilobyte range, as compression typically does not work well for small inputs. Techniques such as variable-length encoding also incur a significant performance penalty which is magnified by large inputs.

LZ4 does not work well with either Kryo configuration as both compression time and size of compressed data are considerably larger than with Java serialization and MessagePack. This behavior happens consistently in all test cases. Inspection of the source code reveals that this is caused by rather unfortunate behavior in Kryo. When Kryo's internal buffer where data is serialized to becomes full, the bytes are written to the underlying `OutputStream`. Kryo also flushes the underlying stream which in this case is the compression stream.

When the compression stream is flushed it must compress and write any buffered bytes. Therefore the size of the compressor's buffer is effectively determined by Kryo. Default buffer size of LZ4 is 4 MB which is 1000 times larger than Kryo's 4 kB buffer. Small amounts of data compress less efficiently and constant flushing also adds other overhead. This also explains

why decompression performance is largely unaffected. The phenomenon is also visible in Zstandard's results although not as severely.

If flushing was avoided or buffer was sized accordingly, LZ4 would likely perform as well with Kryo as with Java serializer and MessagePack. However, the reason for bad interaction of LZ4 and Kryo was found rather late. Moreover, it was also a conscious decision during experiment planning to make only a minimal number of configuration changes to keep the results comparable. Optimizing configuration of some components would give them an unfair advantage and finding optimal configurations for all combinations would be an overwhelming task.

Size-wise Zstandard and Deflate produce very similar results although Zstandard is marginally better. However, Deflate is four to six times slower on compression and up to two times slower on decompression. These results are in line with results reported on Zstandard homepage [61] and show rather clearly that there is no real reason to use Deflate, at least in this use case and perhaps also more generally. Other compressors have clearer tradeoffs: XZ offers very strong compression, LZ4 is very fast and Zstandard is somewhere in the middle.

8.1.4 Throughput performance

In the throughput benchmark, which simulates production-like conditions, differences between serializers are small. Largest differences are seen in the 100 MB case where Kryo (unsafe) is 9 % faster than Java serializer but even this translates to only 630 ms difference. The differences diminish with larger inputs and Java serializer is, in fact, the fastest in the 10 GB case. It would be possible to use different serializer for different sized change cargos. However, differences in runtime are small enough to the point where other than purely performance-related questions start to become relevant. For example, dynamically selecting serializer adds complexity to the codebase as does the use of an external serialization library.

Using compression yields worse throughput in most cases. The only exception is LZ4 in the 100 MB case but again the difference is small enough that it is not meaningful in practice. Kryo's poor interaction with LZ4 is also clearly visible in throughput benchmark although the impact is less severe than in the synthetic benchmarks. However, with Java serializer and MessagePack LZ4 appears to have a good balance between performance and compression ratio as the runtime is very close to the uncompressed case with all input sizes. In case of more constrained data transfers, LZ4 would likely achieve better throughput compared to no compression although it is difficult to say what exactly is the tipping point.

In the tested system serialization, transmission, and deserialization (along with possible (de)compression) happen in parallel so throughput of the whole system is determined by the throughput of the slowest part. Based on synthetic benchmarks, all serializers are capable of serializing and deserializing data at the rate of at least 800 MB/s. As the theoretical throughput of the network is only 125 MB/s, all serializers are able to saturate the network. This could explain why all serializers have very similar performance.

However, the network might not be the main limiting factor. For 10 GB input, the time it takes to serialize and transmit all data varies between 237 s to 247 s for all serializers. Theoretically, the 10 GB payload could be transferred in about 80 seconds so the actual throughput is lower than theoretical by a factor of 3. For smaller inputs, this factor is closer to four. It is unlikely that, for example, chunking alone would add this much overhead. More likely culprits, or at least contributing factors, are gRPC and queuing although identifying the exact bottleneck would require more detailed investigation. Regardless, the results of the throughput benchmark highlight the fact that serialization performance is not all that important if transfer throughput is limited.

What is more, the results confirm an old adage about not trusting synthetic benchmarks. For example, Kryo (unsafe) could have been expected to provide tens of percents better throughput compared to Java serializer. In practice, the difference was less than 10 % even in the best case. Also, in general, all compressors had better throughput than could be expected based on the synthetic benchmarks. For example, in the serialization benchmark XZ compression increases runtime by up to 65 times compared to baseline while in the throughput benchmark the increase is at most 2.4 times. Of course, compression trades time to space but as discussed earlier, it might be difficult to incorporate this into throughput estimates. Despite this the results of the synthetic benchmarks are still valid, one must simply be careful when interpreting them.

One thing that is clear from the results of the throughput benchmark and which is confirmed by the results of the synthetic benchmarks is that the performance of serializers stays quite predictable as the input size grows to gigabyte-range. This is an important conclusion because any future benchmarking can be done using smaller inputs which is considerably faster. Also, although many benchmarks of big data applications no doubt use large inputs, we were not able to find a study which would explicitly test serialization performance with this large inputs or conclude that the input size does not affect the predictability of performance.

Although the experiments were done in the context of a specific system and setup, the results should be applicable more generally. The exact nume-

ric results will vary depending on many factors but it was clearly established that Kryo can significantly improve serialization performance compared to Java serializer, even with very large inputs. Similarly, LZ4 was found to be promising for situations where real-time compression is required. However, the results of this study do not completely agree with the conclusions of other studies which benchmarked data serialization and transfer over the network. Research of Sikdar et al. [53] and Raasveldt et al. [51] found that the performance of the serializer is more important for throughput than the size of the data. In this study, differences between serializers were rather minimal regardless of serialized data size, even though differences in synthetic benchmarks were rather remarkable. On the other hand, the results of this study agree with Raasveldt et al. [51] in that even lightest-weight compression methods are too CPU intensive to improve throughput unless network conditions are sufficiently bad.

The throughput benchmarks were run in a setup with 1 Gbit/s network which is considerably less than 20 Gbit/s which is planned for production setups. To give some scale for these numbers, let us contrast them to the offering of two large public cloud providers: Amazon Web Services (AWS) and Microsoft Azure. AWS promises rather vaguely "up to 10 Gbit/s" network performance for many instance types [3]. If we examine only instances with at least 32 GB of memory (which would be unusually low for the system considered), all such instances can sustain at least 1 Gbit/s transfer speed according to some informal benchmarks [18]. Many instances have higher sustained or burst speed. On the other hand, Azure promises "expected network bandwidth" of at least 1000 Mbit/s for all instances [32]. We can realistically expect 1 Gbit/s network to be available for these types of workloads and in fact, it is even in the lower end of the spectrum. Ousterhout et al. [43] made similar conclusion in their study on Apache Spark performance.

8.2 Future work

Purpose of this study was to compare the performance of Java serialization and compression methods and to identify which serializer-compressor combination yields the best throughput in the presented use case. Libraries were tested mostly with default settings to keep the number of combinations manageable. However, as was seen with Kryo and LZ4, it would be beneficial to explore and fine-tune settings of the most promising libraries and serializer-compressor combinations.

The network is a major factor in the overall throughput. As bandwidth decreases, compression becomes more and more effective. Although it is un-

clear at what amount of bandwidth compression would be always beneficial, there were already some instances where compression improves throughput. In the test setup, there was considerably less bandwidth than there will be in production setup so it is questionable how useful it would be to redo benchmarks with more constrained bandwidth. Evidence also suggests that the amount of bandwidth in the test setup is not unusually high for the tested workloads. On the other hand, multiple worker nodes committing simultaneously might burden the network in such a way that compression would be beneficial, even if there is more bandwidth available than in the test setup.

Raasveldt et al. [51] investigated client protocols of commonly used DBMSs, such as PostgreSQL, and all were found to be highly susceptible to latency. The reason is that all tested protocols run over TCP which requires an acknowledgment for each packet. Therefore it is expected that similar effects would be seen also in the system considered in this thesis. Currently, the system is assumed to be deployed only in single-DC (data center) setups where node-to-node latencies are very low. However, the use of compression should be re-evaluated if multi-DC setups are considered in the future. Also, in such setups there would be likely less bandwidth available.

Finally, in its current form, the system does serialization and compression using a single thread only. It would be possible to split the object graph into smaller pieces and do serialization in parallel in exchange for an increase in code complexity. Although parallel serialization is unlikely to affect the choice of serializer, such setup could favor stronger compression, or any compression to begin with, as compression is quite CPU intensive. However, that should be evaluated if such parallelization is implemented. Another alternative regarding compression is to use a specialized algorithm to compress integers which comprise the bulk of the data. Effectiveness of such algorithms depends largely on the distribution of values but in some cases, they may perform better than general-purpose compression methods [51].

Chapter 9

Conclusions

In this thesis, we evaluated Java serializers and compression methods in the context of a distributed database. The database in question uses object serialization as part of remote commit protocol and has high performance requirements.

A total of three different serializers and four compressors were selected for evaluation. These technologies were assessed using both synthetic benchmarks, which are easy to understand and perform, and a throughput benchmark simulating realistic use of the system. The technologies were compared against the solution already used by the system.

Synthetic benchmarks revealed large differences between selected technologies. Based on these benchmarks, Kryo serializer and LZ4 compressor could be identified as promising. However, good performance in the synthetic benchmarks did not show clearly in the results of the throughput benchmark. Differences between serializers were small enough to be irrelevant and the use of compression was detrimental for throughput. The first result was attributed to data transfer rate being too low, the second result to the data transfer rate being high enough. Based on these conclusions the system continues to use the existing solution.

Although this thesis did not yield immediate improvements in the system, some promising technologies were identified. Furthermore, some possible performance bottlenecks were discovered in the system during the analysis of the results. When these have been investigated and fixed, and the system is overall improved further, use of the aforementioned promising technologies should be re-evaluated. Finally, we confirmed a piece of common wisdom: one should not make choices regarding performance based on intuition but only on cold, hard numbers.

Bibliography

[1] 7z Format. URL: https://www.7-zip.org/7z.html. [Accessed 2019-05-16].

[2] S.P. Ahuja and R. Quintao. "Performance evaluation of Java RMI: A distributed object architecture for internet based applications". In: *Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS* 2000-January (2000), pp. 565–569. ISSN: 15267539. DOI: 10.1109/MASCOT.2000.876585.

[3] Amazon EC2 Instance Types. URL: https://aws.amazon.com/ec2/instance-types/. [Accessed 2019-06-25].

[4] Apache Arrow Homepage. URL: https://arrow.apache.org/. [Accessed 2019-05-14].

[5] Apache Avro 1.9.0 Documentation. URL: http://avro.apache.org/docs/current/. [Accessed 2019-06-26].

[6] Apache Flink: Stateful Computations over Data Streams. URL: https://flink.apache.org/. [Accessed 2019-07-15].

[7] Apache Helix. URL: https://helix.apache.org/. [Accessed 2019-07-21].

[8] Apache Spark - Unified Analytics Engine for Big Data. URL: https://spark.apache.org/. [Accessed 2019-05-13].

[9] Apache ZooKeeper. URL: https://zookeeper.apache.org/. [Accessed 2019-07-21].

[10] K. Beineke, S. Nothaas, and M. Schottner. "Efficient messaging for Java applications running in data centers". In: *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018* (2018), pp. 589–598. DOI: 10.1109/CCGRID.2018.00090.

[11] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. Mckinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanoví, T. VanDrunen, D. Von Dincklage, and B. Wiedermann. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA* 2006 (2006), pp. 169–190. DOI: 10.1145/1167473.1167488.

[12] F. Breg and C.D. Polychronopoulos. "Java virtual machine support for object serialization". In: *ACM 2001 Java Grande/ISCOPE Conference* (2001), pp. 173–180.

[13] Configuration - Spark 2.4.3 Documentation. URL: https://spark.apache.org/docs/latest/configuration.html. [Accessed 2019-05-16].

[14] CVE-2013-0156. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0156. [Accessed 2019-05-10].

[15] P. Deutsch. RFC1951: DEFLATE compressed data format specification version 1.3. 1996.

[16] P. Deutsch and J.-L. Gailly. RFC1950: Zlib compressed data format specification version 3.3. 1996.

[17] Thomas J. Watson IBM Research Center. Research Division, B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, and B. W. Wade. "Notes on distributed databases". 1979.

[18] EC2 Network Performance Cheat Sheet. URL: https://cloudonaut.io/ec2-network-performance-cheat-sheet/. [Accessed 2019-06-25].

[19] M. Falck and M. Nikula. "Big Data – Big Talk or Big Results?" In: (). URL: https://www.relexsolutions.com/big-data-big-talk-or-big-results/. [Accessed 2019-06-03].

[20] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically rigorous Java performance evaluation". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA* (2007), pp. 57–76. DOI: 10.1145/1297027.1297033.

[21] S. Gilbert and N. Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389.

[22] M. Gligoric, D. Marinov, and S. Kamin. "CoDeSe: Fast deserialization via code generation". In: *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings* (2011), pp. 298–308. DOI: 10.1145/2001420.2001456.

[23] gRPC. URL: https://grpc.io/. [Accessed 2019-06-04].

[24] G. Horváth, N. Pataki, and M. Balassi. "Code generation in serializers and comparators of Apache Flink". In: *Proceedings of the 12th Workshop on Implementation, Compilation and Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS 2017* (2017). DOI: 10.1145/3098572.3098579.

[25] jackson-databind. URL: https://github.com/FasterXML/jackson-databind. [Accessed 2019-05-15].

[26] JEP 260: Encapsulate Most Internal APIs. URL: http://openjdk.java.net/jeps/260. [Accessed 2019-05-15].

[27] JSON Schema. URL: https://json-schema.org/. [Accessed 2019-06-26].

[28] M. Kleppmann. "Designing Data-Intensive Applications". O'Reilly Media, Inc., 2017. ISBN: 9781449373320.

[29] Kryo. URL: https://github.com/EsotericSoftware/kryo. [Accessed 2019-05-15].

[30] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. "SparkBench: a Spark benchmarking suite characterizing large-scale in-memory data analytics". In: *Cluster Computing* 20.3 (2017), pp. 2575–2589. ISSN: 13867857. DOI: 10.1007/s10586-016-0723-1.

[31] Linkerd. URL: https://linkerd.io/. [Accessed 2019-07-21].

[32] Linux VM sizes in Azure. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes?toc=%5C%2fazure%5C%2fvirtual-network%5C%2ftoc.json. [Accessed 2019-06-25].

[33] LZ4 - Extremely fast compression. URL: https://lz4.github.io/lz4/. [Accessed 2019-05-16].

[34] lz4-java. URL: https://github.com/lz4/lz4-java. [Accessed 2019-05-16].

[35] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. "Efficient Java RMI for parallel programming". In: *ACM Transactions on Programming Languages and Systems* 23.6 (2001), pp. 747–775. ISSN: 01640925. DOI: 10.1145/506315.506317.

[36] MessagePack. URL: https://msgpack.org. [Accessed 2019-05-15].

[37] MessagePack Specification. URL: https://github.com/msgpack/msgpack/blob/master/spec.md. [Accessed 2019-05-15].

[38] H. Miller, P. Haller, E. Burmako, and M. Odersky. "Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA* (2013), pp. 183–201. DOI: 10.1145/2509136.2509547.

[39] msgpack-java. URL: https://github.com/msgpack/msgpack-java. [Accessed 2019-05-15].

[40] OpenIO. URL: https://www.openio.io/. [Accessed 2019-07-21].

[41] Oracle. Java Object Serialization Specification. URL: https://docs.oracle.com/en/java/javase/12/docs/specs/serialization/index.html. [Accessed 2019-05-15].

[42] Oracle plans to dump risky Java serialization. URL: https://www.infoworld.com/article/3275924/oracle-plans-to-dump-risky-java-serialization.html. [Accessed 2019-05-10].

[43] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. "Making sense of performance in data analytics frameworks". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015* (2015), pp. 293–307.

[44] M. T. Özsu and P. Valduriez. "Principles of distributed database systems". Springer Science & Business Media, 2011. ISBN: 9781441988331. DOI: 10.1007/978-1-4419-8834-8.

[45] B. Petersen, H. Bindner, S. You, and B. Poulsen. "Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the Internet of Things". In: *Proceedings of Computing Conference 2017* 2018-January (2018), pp. 1339–1346. DOI: 10.1109/SAI.2017.8252264.

[46] M. Philippsen, B. Haumacher, and C. Nester. "More efficient serialization and RMI for Java". In: *Concurrency Practice and Experience* 12.7 (2000), pp. 495–518. ISSN: 10403108. DOI: 10.1002/1096-9128(200005)12:7<495::AID-CPE496>3.0.CO;2-W.

[47] pickle — Python object serialization — Python 3.7.4 documentation. URL: https://docs.python.org/3/library/pickle.html. [Accessed 2019-05-10].

[48] PKWARE. .ZIP File Format Specification version 6.3.6. 2019.

[49] A. Prokopec, G. Duboscq, L. Bulej, D. Simon, A. Rosà, P. Tůma, Y. Zheng, T. Würthinger, D. Leopoldseder, M. Studener, A. Villazón, and W. Binder. "Renaissance: Benchmarking suite for parallel applications on the JVM". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2019), pp. 31–47. DOI: 10.1145/3314221.3314637.

[50] Protocol Buffers. URL: https://developers.google.com/protocol-buffers/. [Accessed 2019-05-13].

[51] M. Raasveldt and H. Mühleisen. "Don't hold my data hostage-A case for client protocol redesign". In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1022–1033. ISSN: 21508097. DOI: 10.14778/3115404.3115408.

[52] Ruby 2.6.3 Marshal Documentation. URL: https://ruby-doc.org/core-2.6.3/Marshal.html. [Accessed 2019-05-10].

[53] S. Sikdar, K. Teymourian, and C. Jermaine. "An experimental comparison of complex object implementations for big data systems". In: *SoCC 2017 - Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 432–444. DOI: 10.1145/3127479.3129248.

[54] SPECjvm2008. URL: https://www.spec.org/jvm2008/. [Accessed 2019-07-20].

[55] J.D. Ullman and J. Widom. "A First Course in Database Systems, Third Edition". Pearson Education, Inc., 2008. ISBN: 9780135021767.

[56] M. Van Steen and A.S. Tanenbaum. "Distributed Systems, Third Edition". Van Steen, M., 2017. ISBN: 9789081540629.

[57] J. Vanura and P. Kriz. "Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10969 LNCS (2018), pp. 166–175. ISSN: 03029743. DOI: 10.1007/978-3-319-94376-3_11.

[58] W3C XML Schema. URL: https://www.w3.org/XML/Schema. [Accessed 2019-06-26].

[59] XZ for Java. URL: https://tukaani.org/xz/java.html. [Accessed 2019-05-16].

[60] ysoserial. URL: https://github.com/frohoff/ysoserial. [Accessed 2019-05-10].

[61] Zstandard. URL: `https://facebook.github.io/zstd/`. [Accessed 2019-05-16].

[62] zstd-jni. URL: `https://github.com/luben/zstd-jni`. [Accessed 2019-05-16].

Appendix A

Serialization benchmark results

Results of the serialization benchmark are listed in this appendix. The tables contain average runtime for each serializer-compressor combination and also relative score and absolute runtime difference compared to the baseline, Java serializer without compression. The score is calculated by dividing the average runtime of the combination by the average runtime of the baseline.

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	0.108	1.00	0.
Java	LZ4	0.345	3.19	0.237
Java	Zstandard	0.825	7.62	0.717
Java	Deflate	3.46	32.0	3.35
Java	XZ	3.93	36.3	3.82
Kryo (default)	None	0.114	1.05	0.005 91
Kryo (default)	LZ4	1.12	10.4	1.01
Kryo (default)	Zstandard	1.10	10.2	0.992
Kryo (default)	Deflate	3.19	29.4	3.08
Kryo (default)	XZ	3.93	36.3	3.82
Kryo (unsafe)	None	0.0367	0.339	−0.0716
Kryo (unsafe)	LZ4	1.15	10.6	1.04
Kryo (unsafe)	Zstandard	0.889	8.21	0.781
Kryo (unsafe)	Deflate	3.23	29.8	3.12
Kryo (unsafe)	XZ	4.00	36.9	3.89
MessagePack	None	0.0654	0.604	−0.0429
MessagePack	LZ4	0.292	2.70	0.184
MessagePack	Zstandard	0.727	6.71	0.619
MessagePack	Deflate	3.02	27.9	2.91
MessagePack	XZ	3.67	33.9	3.56

Table A.1: Serialization runtime (100 MB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	0.778	1.00	0.
Java	LZ4	2.44	3.13	1.66
Java	Zstandard	5.87	7.55	5.09
Java	Deflate	30.1	38.7	29.3
Java	XZ	35.1	45.1	34.3
Kryo (default)	None	1.12	1.45	0.346
Kryo (default)	LZ4	8.50	10.9	7.72
Kryo (default)	Zstandard	8.80	11.3	8.03
Kryo (default)	Deflate	29.0	37.2	28.2
Kryo (default)	XZ	35.9	46.1	35.1
Kryo (unsafe)	None	0.261	0.336	−0.516
Kryo (unsafe)	LZ4	9.00	11.6	8.22
Kryo (unsafe)	Zstandard	6.27	8.06	5.49
Kryo (unsafe)	Deflate	30.0	38.5	29.2
Kryo (unsafe)	XZ	37.4	48.1	36.6
MessagePack	None	0.877	1.13	0.0994
MessagePack	LZ4	2.47	3.17	1.69
MessagePack	Zstandard	5.18	6.66	4.40
MessagePack	Deflate	28.2	36.3	27.5
MessagePack	XZ	34.5	44.3	33.7

Table A.2: Serialization runtime (1 GB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	7.36	1.00	0.
Java	LZ4	22.2	3.02	14.8
Java	Zstandard	53.3	7.24	45.9
Java	Deflate	304	41.3	297
Java	XZ	377	51.2	370
Kryo (default)	None	10.8	1.47	3.47
Kryo (default)	LZ4	83.3	11.3	75.9
Kryo (default)	Zstandard	84.3	11.4	76.9
Kryo (default)	Deflate	285	38.7	278
Kryo (default)	XZ	368	49.9	360
Kryo (unsafe)	None	5.82	0.790	−1.55
Kryo (unsafe)	LZ4	89.6	12.2	82.2
Kryo (unsafe)	Zstandard	61.4	8.34	54.0
Kryo (unsafe)	Deflate	296	40.2	289
Kryo (unsafe)	XZ	376	51.0	368
MessagePack	None	7.72	1.05	0.359
MessagePack	LZ4	22.8	3.09	15.4
MessagePack	Zstandard	51.1	6.94	43.8
MessagePack	Deflate	272	36.9	265
MessagePack	XZ	360	48.8	352

Table A.3: Serialization runtime (10 GB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	58.6	1.00	0.
Java	LZ4	203	3.46	144
Java	Zstandard	525	8.95	466
Java	Deflate	3000	51.2	2940
Java	XZ	3830	65.4	3770
Kryo (default)	None	98.8	1.69	40.2
Kryo (default)	LZ4	826	14.1	767
Kryo (default)	Zstandard	826	14.1	767
Kryo (default)	Deflate	2820	48.1	2760
Kryo (default)	XZ	3590	61.2	3530
Kryo (unsafe)	None	43.0	0.733	−15.7
Kryo (unsafe)	LZ4	880	15.0	821
Kryo (unsafe)	Zstandard	802	13.7	744
Kryo (unsafe)	Deflate	3020	51.5	2960
Kryo (unsafe)	XZ	3780	64.5	3720
MessagePack	None	85.4	1.46	26.8
MessagePack	LZ4	233	3.97	174
MessagePack	Zstandard	528	9.02	470
MessagePack	Deflate	2800	47.8	2740
MessagePack	XZ	3510	59.8	3450

Table A.4: Serialization runtime (100 GB)

Appendix B

Deserialization benchmark results

Results of the deserialization benchmark are listed in this appendix. The tables contain average runtime for each serializer-compressor combination and also relative score and absolute runtime difference compared to the baseline, Java serializer without compression. The score is calculated by dividing the average runtime of the combination by the average runtime of the baseline.

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	0.122	1.00	0.
Java	LZ4	0.150	1.24	0.0288
Java	Zstandard	1.55	12.8	1.43
Java	Deflate	1.08	8.84	0.954
Java	XZ	1.60	13.2	1.48
Kryo (default)	None	0.114	0.937	−0.007 69
Kryo (default)	LZ4	0.138	1.14	0.0165
Kryo (default)	Zstandard	0.541	4.45	0.419
Kryo (default)	Deflate	0.694	5.70	0.572
Kryo (default)	XZ	2.61	21.5	2.49
Kryo (unsafe)	None	0.0515	0.423	−0.0702
Kryo (unsafe)	LZ4	0.0917	0.754	−0.0299
Kryo (unsafe)	Zstandard	0.260	2.13	0.138
Kryo (unsafe)	Deflate	0.703	5.78	0.581
Kryo (unsafe)	XZ	2.57	21.2	2.45
MessagePack	None	0.0772	0.635	−0.0445
MessagePack	LZ4	0.128	1.05	0.006 47
MessagePack	Zstandard	0.314	2.58	0.192
MessagePack	Deflate	0.621	5.10	0.499
MessagePack	XZ	2.66	21.9	2.54

Table B.1: Deserialization runtime (100 MB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	1.07	1.00	0.
Java	LZ4	1.17	1.10	0.106
Java	Zstandard	4.60	4.31	3.53
Java	Deflate	5.40	5.06	4.33
Java	XZ	13.1	12.3	12.1
Kryo (default)	None	1.18	1.11	0.115
Kryo (default)	LZ4	1.47	1.38	0.402
Kryo (default)	Zstandard	3.77	3.53	2.70
Kryo (default)	Deflate	4.20	3.93	3.13
Kryo (default)	XZ	23.9	22.4	22.8
Kryo (unsafe)	None	0.501	0.469	−0.566
Kryo (unsafe)	LZ4	1.02	0.953	−0.0507
Kryo (unsafe)	Zstandard	2.67	2.50	1.60
Kryo (unsafe)	Deflate	4.21	3.95	3.15
Kryo (unsafe)	XZ	23.6	22.2	22.6
MessagePack	None	1.07	1.01	0.007 41
MessagePack	LZ4	1.28	1.20	0.216
MessagePack	Zstandard	2.68	2.51	1.61
MessagePack	Deflate	3.95	3.70	2.88
MessagePack	XZ	23.6	22.1	22.6

Table B.2: Deserialization runtime (1 GB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	10.5	1.00	0.
Java	LZ4	13.8	1.31	3.28
Java	Zstandard	34.3	3.27	23.8
Java	Deflate	51.0	4.86	40.5
Java	XZ	119	11.4	109
Kryo (default)	None	12.1	1.15	1.62
Kryo (default)	LZ4	13.1	1.25	2.64
Kryo (default)	Zstandard	36.9	3.52	26.5
Kryo (default)	Deflate	41.8	3.98	31.3
Kryo (default)	XZ	225	21.4	214
Kryo (unsafe)	None	8.66	0.825	−1.83
Kryo (unsafe)	LZ4	11.0	1.05	0.485
Kryo (unsafe)	Zstandard	28.1	2.67	17.6
Kryo (unsafe)	Deflate	42.1	4.01	31.6
Kryo (unsafe)	XZ	221	21.1	210
MessagePack	None	11.8	1.12	1.28
MessagePack	LZ4	16.0	1.53	5.51
MessagePack	Zstandard	28.9	2.76	18.4
MessagePack	Deflate	39.8	3.79	29.3
MessagePack	XZ	221	21.0	210

Table B.3: Deserialization runtime (10 GB)

Serializer	Compressor	Average runtime (s)	Score	Difference (s)
Java	None	105	1.00	0.
Java	LZ4	130	1.25	25.6
Java	Zstandard	333	3.18	228
Java	Deflate	506	4.84	402
Java	XZ	1470	14.1	1370
Kryo (default)	None	123	1.17	18.2
Kryo (default)	LZ4	143	1.37	38.8
Kryo (default)	Zstandard	382	3.65	277
Kryo (default)	Deflate	433	4.14	329
Kryo (default)	XZ	2350	22.5	2250
Kryo (unsafe)	None	80.3	0.768	−24.3
Kryo (unsafe)	LZ4	107	1.02	2.07
Kryo (unsafe)	Zstandard	369	3.53	264
Kryo (unsafe)	Deflate	447	4.28	343
Kryo (unsafe)	XZ	2380	22.7	2270
MessagePack	None	112	1.07	7.24
MessagePack	LZ4	139	1.33	34.1
MessagePack	Zstandard	272	2.60	167
MessagePack	Deflate	427	4.08	322
MessagePack	XZ	2330	22.2	2220

Table B.4: Deserialization runtime (100 GB)

Appendix C

Size of serialized data

Amount of serialized (and compressed) data produced by each serializer-compressor combination is listed in this appendix. The tables also contain relative score and absolute size difference compared to the baseline, Java serializer without compression. The score is calculated by dividing the data size of the combination by the data size of the baseline. Additionally, the tables also include compression ratio calculated as

$$\text{Compression ratio} = \frac{\text{Uncompressed size}}{\text{Compressed size}}$$

Compression ratios are calculated separately for each serializer.

Serializer	Compressor	Serialized data (MB)	Compression ratio	Score	Difference (MB)
Java	None	108	1.00	1.00	0.
Java	LZ4	22.2	4.86	0.206	−85.5
Java	Zstandard	11.4	9.43	0.106	−96.3
Java	Deflate	12.0	8.97	0.111	−95.7
Java	XZ	7.21	14.9	0.0670	−100
Kryo (default)	None	90.8	1.00	0.843	−16.9
Kryo (default)	LZ4	24.8	3.66	0.230	−82.9
Kryo (default)	Zstandard	12.5	7.26	0.116	−95.2
Kryo (default)	Deflate	12.0	7.60	0.111	−95.7
Kryo (default)	XZ	7.90	11.5	0.0733	−99.8
Kryo (unsafe)	None	107	1.00	0.993	−0.716
Kryo (unsafe)	LZ4	25.2	4.25	0.234	−82.5
Kryo (unsafe)	Zstandard	13.0	8.21	0.121	−94.7
Kryo (unsafe)	Deflate	12.0	8.95	0.111	−95.7
Kryo (unsafe)	XZ	7.59	14.1	0.0705	−100
MessagePack	None	92.3	1.00	0.857	−15.4
MessagePack	LZ4	22.0	4.20	0.204	−85.7
MessagePack	Zstandard	11.3	8.15	0.105	−96.4
MessagePack	Deflate	11.9	7.73	0.111	−95.8
MessagePack	XZ	7.52	12.3	0.0698	−100

Table C.1: Size of serialized data (100 MB)

Serializer	Compressor	Serialized data (MB)	Compression ratio	Score	Difference (MB)
Java	None	1020	1.00	1.00	0.
Java	LZ4	219	4.67	0.214	−805
Java	Zstandard	112	9.15	0.109	−913
Java	Deflate	118	8.65	0.116	−906
Java	XZ	71.4	14.4	0.0696	−953
Kryo (default)	None	877	1.00	0.856	−148
Kryo (default)	LZ4	246	3.56	0.240	−779
Kryo (default)	Zstandard	123	7.10	0.120	−901
Kryo (default)	Deflate	118	7.41	0.115	−906
Kryo (default)	XZ	78.2	11.2	0.0763	−947
Kryo (unsafe)	None	1020	1.00	0.999	−1.46
Kryo (unsafe)	LZ4	249	4.11	0.243	−776
Kryo (unsafe)	Zstandard	128	8.00	0.125	−897
Kryo (unsafe)	Deflate	118	8.64	0.116	−906
Kryo (unsafe)	XZ	75.3	13.6	0.0735	−949
MessagePack	None	892	1.00	0.871	−132
MessagePack	LZ4	217	4.10	0.212	−807
MessagePack	Zstandard	111	8.04	0.108	−914
MessagePack	Deflate	118	7.57	0.115	−907
MessagePack	XZ	74.5	12.0	0.0727	−950

Table C.2: Size of serialized data (1 GB)

Serializer	Compressor	Serialized data (MB)	Compression ratio	Score	Difference (MB)
Java	None	10 100	1.00	1.00	0.
Java	LZ4	2190	4.61	0.217	−7900
Java	Zstandard	1090	9.21	0.109	−8990
Java	Deflate	1180	8.53	0.117	−8900
Java	XZ	710	14.2	0.0704	−9370
Kryo (default)	None	8750	1.00	0.867	−1340
Kryo (default)	LZ4	2460	3.56	0.244	−7630
Kryo (default)	Zstandard	1210	7.25	0.120	−8880
Kryo (default)	Deflate	1180	7.41	0.117	−8900
Kryo (default)	XZ	778	11.2	0.0772	−9310
Kryo (unsafe)	None	10 100	1.00	1.000	−1.54
Kryo (unsafe)	LZ4	2490	4.05	0.247	−7600
Kryo (unsafe)	Zstandard	1250	8.09	0.124	−8840
Kryo (unsafe)	Deflate	1180	8.53	0.117	−8900
Kryo (unsafe)	XZ	749	13.5	0.0742	−9340
MessagePack	None	8810	1.00	0.874	−1270
MessagePack	LZ4	2170	4.06	0.215	−7910
MessagePack	Zstandard	1080	8.13	0.108	−9000
MessagePack	Deflate	1180	7.49	0.117	−8910
MessagePack	XZ	741	11.9	0.0735	−9340

Table C.3: Size of serialized data (10 GB)

Serializer	Compressor	Serialized data (MB)	Compression ratio	Score	Difference (MB)
Java	None	101 000	1.00	1.00	0.
Java	LZ4	21 900	4.61	0.217	−78 900
Java	Zstandard	11 000	9.13	0.110	−89 800
Java	Deflate	11 800	8.54	0.117	−89 000
Java	XZ	7130	14.1	0.0707	−93 700
Kryo (default)	None	87 600	1.00	0.869	−13 200
Kryo (default)	LZ4	24 600	3.56	0.244	−76 200
Kryo (default)	Zstandard	12 200	7.17	0.121	−88 600
Kryo (default)	Deflate	11 800	7.43	0.117	−89 000
Kryo (default)	XZ	7810	11.2	0.0775	−93 000
Kryo (unsafe)	None	101 000	1.00	1.000	−2.50
Kryo (unsafe)	LZ4	24 900	4.05	0.247	−75 900
Kryo (unsafe)	Zstandard	12 200	8.25	0.121	−88 600
Kryo (unsafe)	Deflate	11 800	8.55	0.117	−89 000
Kryo (unsafe)	XZ	7510	13.4	0.0745	−93 300
MessagePack	None	88 100	1.00	0.874	−12 700
MessagePack	LZ4	21 700	4.06	0.215	−79 100
MessagePack	Zstandard	10 900	8.05	0.109	−89 900
MessagePack	Deflate	11 700	7.50	0.117	−89 100
MessagePack	XZ	7440	11.8	0.0738	−93 400

Table C.4: Size of serialized data (100 GB)

Appendix D

Throughput benchmark results

Results of the throughput benchmark are listed in this appendix. The tables contain average runtime for each serializer-compressor combination and also relative score and absolute runtime difference compared to the baseline, Java serializer without compression. The score is calculated by dividing the average runtime of the combination by the average runtime of the baseline. Additionally, value $S+Q$ in each table is the average time taken to serialize change cargo and drain the chunk queue.

Serializer	Compressor	Average runtime (s)	S+Q (s)	Score	Difference (s)
Java	None	6.92	3.13	1.00	0.
Java	LZ4	6.47	2.81	0.934	−0.456
Java	Zstandard	6.92	3.26	0.999	−0.005 20
Java	Deflate	10.4	6.64	1.50	3.47
Java	XZ	11.0	7.18	1.60	4.12
Kryo (default)	None	6.67	3.16	0.963	−0.257
Kryo (default)	LZ4	7.35	3.50	1.06	0.423
Kryo (default)	Zstandard	7.30	3.54	1.05	0.374
Kryo (default)	Deflate	9.95	6.12	1.44	3.03
Kryo (default)	XZ	10.8	7.15	1.56	3.89
Kryo (unsafe)	None	6.30	2.82	0.910	−0.626
Kryo (unsafe)	LZ4	7.57	3.85	1.09	0.647
Kryo (unsafe)	Zstandard	7.34	3.12	1.06	0.417
Kryo (unsafe)	Deflate	9.36	5.86	1.35	2.44
Kryo (unsafe)	XZ	10.5	6.89	1.51	3.57
MessagePack	None	6.81	3.11	0.984	−0.114
MessagePack	LZ4	6.61	2.88	0.954	−0.315
MessagePack	Zstandard	7.27	3.28	1.05	0.343
MessagePack	Deflate	9.70	5.97	1.40	2.78
MessagePack	XZ	10.4	6.66	1.50	3.45

Table D.1: Throughput runtime (100 MB)

Serializer	Compressor	Average runtime (s)	S+Q (s)	Score	Difference (s)
Java	None	44.5	26.5	1.00	0.
Java	LZ4	45.1	27.5	1.01	0.657
Java	Zstandard	49.5	31.7	1.11	5.04
Java	Deflate	76.8	59.0	1.73	32.3
Java	XZ	83.6	65.8	1.88	39.2
Kryo (default)	None	43.6	25.8	0.982	−0.820
Kryo (default)	LZ4	51.5	33.4	1.16	7.06
Kryo (default)	Zstandard	50.2	32.2	1.13	5.77
Kryo (default)	Deflate	75.9	58.1	1.71	31.5
Kryo (default)	XZ	79.8	61.9	1.79	35.3
Kryo (unsafe)	None	44.4	26.5	0.999	−0.0481
Kryo (unsafe)	LZ4	51.9	33.7	1.17	7.39
Kryo (unsafe)	Zstandard	47.9	29.7	1.08	3.40
Kryo (unsafe)	Deflate	74.8	56.7	1.68	30.3
Kryo (unsafe)	XZ	81.9	63.7	1.84	37.4
MessagePack	None	46.7	28.9	1.05	2.27
MessagePack	LZ4	45.8	27.5	1.03	1.35
MessagePack	Zstandard	47.7	29.5	1.07	3.26
MessagePack	Deflate	73.5	55.3	1.65	29.1
MessagePack	XZ	79.6	61.1	1.79	35.1

Table D.2: Throughput runtime (1 GB)

Serializer	Compressor	Average runtime (s)	S+Q (s)	Score	Difference (s)
Java	None	294	238	1.00	0.
Java	LZ4	308	247	1.05	13.6
Java	Zstandard	363	302	1.23	68.2
Java	Deflate	626	569	2.13	332
Java	XZ	704	644	2.39	409
Kryo (default)	None	305	247	1.03	10.2
Kryo (default)	LZ4	364	304	1.24	69.6
Kryo (default)	Zstandard	375	314	1.27	80.9
Kryo (default)	Deflate	578	522	1.96	284
Kryo (default)	XZ	669	608	2.27	374
Kryo (unsafe)	None	297	238	1.01	2.67
Kryo (unsafe)	LZ4	390	330	1.32	95.5
Kryo (unsafe)	Zstandard	366	305	1.24	71.4
Kryo (unsafe)	Deflate	629	572	2.14	335
Kryo (unsafe)	XZ	700	638	2.38	405
MessagePack	None	298	242	1.01	3.97
MessagePack	LZ4	320	259	1.09	25.3
MessagePack	Zstandard	348	285	1.18	53.2
MessagePack	Deflate	602	543	2.04	307
MessagePack	XZ	671	609	2.28	376

Table D.3: Throughput runtime (10 GB)